Uncle Maker: (Time)Stamping Out The Competition in Ethereum

AVIV YAISH, The Hebrew University, Israel
GILAD STERN, The Hebrew University, Israel
AVIV ZOHAR, The Hebrew University, Israel

We present an attack on Ethereum's consensus mechanism which can be used by miners to obtain consistently higher mining rewards compared to the honest protocol. This attack is novel in that it does not entail withholding blocks or any behavior which has a non-zero probability of earning less than mining honestly, in contrast with the existing literature.

This risk-less attack relies instead on manipulating block timestamps, and carefully choosing whether and when to do so. We present this attack as an algorithm, which we then analyze to evaluate the revenue a miner obtains from it, and its effect on a miner's absolute and relative share of the main-chain blocks.

The attack allows an attacker to replace competitors' main-chain blocks after the fact with a block of its own, thus causing the replaced block's miner to lose all fees for the transactions contained within the block, which will be demoted from the main-chain. This block, although "kicked-out" of the main-chain, will still be eligible to be referred to by other main-chain blocks, thus becoming what is commonly called in Ethereum an *uncle*.

We proceed by defining multiple variants of this attack, and assessing whether any of these attacks has been performed in the wild. Surprisingly, we find that this is indeed true, making this the first case of a confirmed consensus-level manipulation performed on a major cryptocurrency.

Additionally, we implement a variant of this attack as a patch for Go Ethereum (geth), Ethereum's most popular client, making it the first consensus-level attack on Ethereum which is implemented as a patch. Finally, we suggest concrete fixes for Ethereum's protocol and implemented them as a patch for geth which can be adopted quickly and mitigate the attack and its variants.

CCS Concepts: • **Applied computing** → **Digital cash**; • **Security and privacy** → *Economics of security and privacy*; **Distributed systems security**.

Additional Key Words and Phrases: cryptocurrency, blockchain, proof of work, consensus, security

ACM Reference Format:
Aviv Yaish, Gilad Stern, and Aviv Zohar. 2022. Uncle Maker: (Time)Stamping Out The Competition in Ethereum. 1, 1 (July 2022), 66 pages. https://doi.org/10.13140/RG.2.2.27813.99043

Authors' addresses: Aviv Yaish, aviv.yaish@mail.huji.ac.il, The Hebrew University, Jerusalem, Israel; Gilad Stern, Gilad.Stern@mail.huji.ac.il, The Hebrew University, Jerusalem, Israel; Aviv Zohar, avivz@cs.huji.ac.il, The Hebrew University, Jerusalem, Israel.

https://doi.org/10.13140/RG.2.2.27813.99043

Publication date: July 2022.

1 INTRODUCTION

Cryptocurrencies such as Bitcoin [89] and Ethereum [15] rely on an elaborate incentive system to encourage users to participate in operating the decentralized mechanism which underlies them and maintain the mechanism's integrity in the face of adversaries. Such participants are called *miners*.

Thus, a cryptocurrency's incentive mechanism is inherent to its security. Indeed, among the myriad cryptocurrencies and theoretical protocols which have arrived after Bitcoin, some have dedicated considerable efforts in order to analyze existing mechanisms or design new ones to make sure miners will not have an incentive to foul-play and game the system for their advantage [70, 78, 92, 111, 128].

Ethereum, in particular, is known for adopting changes rapidly, without always carefully examining them and the effect they might have on the incentives of miners [103, 104]. Thus, changes which were designed to mitigate one attack [14], open the door for multiple new ones [98, 127].

On the eve of the transition of Ethereum to a completely new mechanism, also known as *The Merge* [39], we present a new incentive-driven attack on the current Ethereum mechanism. This attack is novel not only in that it relies on the new, unexplored foundations of timestamp manipulations, but also because it is *always* more profitable than following the "honest" rules laid down by the protocol's designers, meaning that our attack strategy strictly dominates the honest strategy.

Previously, Ethereum miners were mostly known to manipulate Ethereum's *application layer*, for example by exploiting vulnerabilities in *smart contracts*, which are applications built on top of blockchains [17], while attacks on the underlying consensus were either entirely theoretical yet practically infeasible, or targeted small implementation bugs which were quickly fixed [5, 127].

In our work, we introduce the first consensus-level attack which is actually feasible to execute. Additionally, we provide the first proof that a consensus-level attack was performed in the wild.

Our Contributions. We make the following contributions:

- We introduce a novel attack vector on proof-of-work (PoW) cryptocurrencies which relies on *timestamp manipulations*, instead of traditional ones such as block withholding [50, 102].
- We provide a thorough description of a concrete attack called the riskless uncle maker (RUM) attack which relies on this new attack technique.
- We prove that the RUM attack is riskless in Ethereum in the sense that an attacker which utilizes it is guaranteed to make at least the same absolute and relative profits when compared to mining according to the honest protocol.
- We describe multiple variants of the attack.
- We investigate the Ethereum blockchain and prove the attack was executed in the wild, thus providing the *first* conclusive evidence of consensus tampering in a major cryptocurrency.
- We have written a fully-functioning implementation of a variant of the attack as a patch for geth, Ethereum's most popular client.
- We provide several techniques to mitigate the attack and its variants, including a patch for geth which prevents the attack altogether.

A summary of our contributions is given in Table 1.

Paper structure. The paper is structured as follows: we begin by going over some implications arising from our work in Section 2. Then, we review the relevant background information required for this work in Section 3, and proceed to describe the paper's model in Section 4. We utilize the model to give an algorithmic description of the attack in Section 5, and then theoretically prove that the attack is riskless and guaranteed to be more profitable than mining honestly in Section 6. We describe the fingerprints which a successful execution of the attack leaves on the blockchain

Table 1. A comparison of our attack and previous ones.

	Uncle Maker	Selfish Mining	Stubborn Mining	Coin Hopping	Energy Equilibria	Stretch & Squeeze
Analyzed on Ethereum	✓	✓	✓	-	-	✓
Doesn't require block withholding	✓	-	-	✓	✓	✓
Always more profitable than mining honestly	✓	-	-	-	-	-
Has a patch for Ethereum clients	✓	-	-	-	-	-

and prove that indeed such attacks have been performed in the wild in Section 7. We suggest mitigations for the attack in Section 8. We review related works in Section 9, and finally conclude in Section 10.

2 IMPLICATIONS

In this section, we would like to discuss some important implications arising from our work.

2.1 Reducing Mechanism Attack Surface

In Ethereum, miners have quite a lot of "wiggle-room" when setting timestamps. Users then partially use their local clock in order to verify them, instead of using some consensus-achieved notion of network time, such as in Bitcoin [114, 125].

We argue that when consensus mechanisms allows such user decisions, their attack surface grows. One natural conclusion is to design mechanisms such that users have the bare minimum of choice in setting certain parameters, like timestamps. If users have some choice, the protocol's designers must make sure that this choice has the minimal possible effect on the protocol's safety.

Timestamps, in particular, affect the difficulty of mining and thus the consensus mechanism. As we will show in our paper, this allows miners to use timestamps which do not correspond to the actual real-world time in order to gain an upper-hand over their competitors.

In spite of this, previous academic research has only made a cursory attempt at examining the potential use of timestamp manipulations to attack the consensus layer of cryptocurrency systems [13, 108, 127], focusing instead on the application layer [2, 17, 24, 84, 105].

2.2 Ethereum-like Mechanisms

In this work, we mainly focus on Ethereum. However, our work has major implications for any mechanism which relies on timestamps in a similar manner.

Among the notable protocols that belong to this category we can list Ethereum Classic (which has publicly committed to continue using the standard Ethereum-like PoW protocol "indefinitely" [19, 20]) and EthereumPoW (a planned fork of Ethereum which is planned to go live when Ethereum finally transitions to its new mechanism [42, 97]).

2.3 Traceable Safety

In this work, we used publicly available on-chain data to retroactively trace an attack on the Ethereum blockchain, making this the first such real-world consensus-level attack discovered on

a major cryptocurrency. This is in spite of previous works attempting to do so, even using very sophisticated means such as setting up multiple nodes in various geographical locations, collecting data and analyzing it [79, 80, 82].

These works have specifically attempted to find instances of block-withholding attacks, which requires being online and finding out about the attacks as they happen. That is to say, block withholding does not leave clear fingerprints in publicly available transcript of the underlying mechanisms, e.g. it is not immediately apparent from the actual blockchain. Thus, such attacks cannot be retroactively traced using on-chain data.

We would like to crystallize the above insight into the following property:

Definition 1 ($\mathcal{A}$-Traceability). *Given a mechanism* Π *and attack on it* $\mathcal{A}$*, we will say* Π *is* $\mathcal{A}$*-traceable if the attack is detectable from the public transcript of* Π*, up to some negligible probability.*

Mechanisms preferably should be designed to make sure there is a public record of potential malfeasance; ideally, to allow post-fact identification of attacks, even those which the mechanism's creators haven't conceived when designing the system. This can be compared to the traditional cryptographic notion of *public verifiability*, which allows actors to check the correctness of protocol transcripts even *after the fact*, and even if they did not originally participate in the protocol. This is in contrast to traditional distributed systems in which real-time participation is required in order to attest to the validity of its execution at the time.

3 BACKGROUND

We will now briefly go over the background knowledge required for understanding our work.

The mechanism utilized by both Ethereum and Bitcoin is called PoW. It requires entities called *miners* to collect user-made *transactions* into *blocks*. As blocks are limited in capacity, users can incentivize miners to prefer their transactions over competing ones by paying them *transaction fees* [27, 76, 83, 101].

In order to create a block which will be considered valid under the PoW mechanism, miners must expend tremendous computational resources to find a solution to a cryptographic puzzle, a process which is also called *mining*. The solution for the puzzle is defined to be an input to a cryptographic hash function such that the corresponding output will be lower than some *target* value. The currently known best method for finding such an input is to perform a brute-force search [26, 69, 112].

Once this solution is found, it is included in the block and broadcasted to the network, with each user validating the solution to ensure it is indeed correct. In order to compensate miners for their efforts, the mechanism allocates a certain amount of tokens to the creator of each valid block. These tokens are commonly called the *block rewards*. Cryptocurrencies commonly pursue a notion of giving miners a *fair* share of the rewards, meaning that the fraction of the all rewards which they earn should be equal to the fraction of computation which they contribute to upholding the underlying mechanism [50, 92].

In order to make sure that the difficulty of the cryptographic puzzle is neither too easy for the network nor does it exceed the computational resources of the network, cryptocurrencies employ a *difficulty-adjustment algorithm* (*DAA*) [23, 127] which periodically attempts to gauge the amount of computational resources currently available on the network and adjust mining difficulty accordingly.

PoW cryptocurrencies are known for their unsteady revenue flow, leading miners to usually team-up and form mining *pools*. Participants in the pool split profits among themselves using a variety of schemes, often-times doing so according to each participant's relative contribution to the pool. This contribution is measured by counting the amount of *shares* which each user submits;

these are blocks which are below the protocol's real difficulty threshold, but above some inner one set by the pool [58, 77, 100, 107].

In cryptocurrencies such as Ethereum, each block must reference at least one previous block, which is called the block's *parent*. Thus, a chain of blocks is formed, also known as a *blockchain*. The first and last blockchain blocks are usually respectively nicknamed the *genesis* and the *tip* of the blockchain, while the currently mined block is commonly called the *pending* block.

As the blockchain contains all user-made transactions, it in essence provides some notion of "history". Thus, the entire network must agree on some specific history in order to facilitate money transfers between users in a way which preserves the system's integrity. A *race* or *tie* occurs if there are multiple possible chains, e.g. when a miner hears of two blocks at the same time. Usually, protocols define a *tie-breaking* rule to pick between the different options. The chain which is picked is referred to as the *main-chain*.

In Ethereum, blocks which are mined after a tie can pick, in addition to a parent, up to two *uncle* blocks. Ethereum's mechanism was designed to take such uncle blocks into consideration when deciding on the network's main-chain in the hopes that doing so would increase the mechanism's security [78]. The resulting structure is no longer a block*chain* but rather a *block directed-acyclic-graph* (*block-DAG*) [78, 109]. Although Ethereum relies on uncle blocks and thus on a block directed-acyclic-graph (block-DAG), for brevity we will refer to the resulting data structure as a block-chain.

In order to participate in the Ethereum protocol, one must use an Ethereum *client*, meaning a program which implements the protocol's rules. The most popular client is *Go Ethereum* (*geth*) [38, 40, 43, 73], which is officially endorsed by the Ethereum Foundation. Communication with a local client and with other nodes is performed using a standardized protocol based on remote procedure calls [122].

For more details on cryptocurrencies see [3, 7, 90, 121].

4 MODEL

We proceed to formally define the model used in the paper.

4.1 Cryptocurrency System

We will begin our model description by giving precise definitions for Ethereum's consensus and reward mechanisms. All notations and acronyms used in this section (and throughout the paper) are detailed in Appendix G.

Block-DAG Structure. In Ethereum, each block must reference a *parent* block, and can reference up to two *uncle* blocks; these are blocks that share a common ancestor with it (up to a depth of 8 blocks), but were not referenced by any main-chain block [35, 125]. Uncles are also sometimes called *ommer* blocks by the Ethereum Foundation [125]. An ommer is the equivalent gender-neutral term for the same familial relation.

Block Timestamps. Ethereum blocks store timestamps in the *UNIX time* format, which measures time by counting the number of seconds elapsed since January 1st, 1970 [66]. Timestamps are saved as integer values and thus do not have a resolution of less than one second. Using these timestamps, miners check the time that has passed between blocks and adjust mining difficulty if the time exceeds some predefined window.

Ethereum enforces relatively strict requirements on timestamps compared to other cryptocurrencies: a newly heard-of block's timestamp cannot be more than 15 seconds in the future when compared to the local clock, and it must be at least one second after its parent's timestamp [29, 35, 125].

Mining Difficulty. Ethereum's difficulty-adjustment algorithm (DAA) strives to keep a rate of one block per $12-14$ seconds [31, 54, 125]. It does so by lowering the difficulty of the block which is *currently* mined if too much time has passed, according to the block's and its parent's timestamps. Ostensibly, the mechanism's designers hoped that this would prevent long stretches of time wherein no new block is mined. Unfortunately, as we will show, such DAAs are susceptible to manipulations.

Formally, each block's difficulty depends on:

- Its parent's difficulty, which we denote by d.
- The timestamp difference from its parent, denoted as t.
- If it references uncles or not: $u \stackrel{\text{def}}{=} 1$ iff has uncles.

Using these, the difficulty of the block is defined as the maximum between 2^{17} and the following term [32, 36, 125]:

$$d + \max\left(1 + u - \lfloor \frac{t}{9} \rfloor, -99\right) \cdot \lfloor \frac{d}{2048} \rfloor \tag{1}$$

For posterity, the difficulty of the genesis block is 2^{34}. As Ethereum's difficulty has skyrocketed compared to 2^{17} and was above 2^{34} since block 15, meaning that the difficulty was higher for at least $15 \cdot 10^6$ blocks. Consequently, we use Eq. (1) as-is for conciseness. Additionally, to facilitate Ethereum's long-delayed migration to a new mechanism which does not rely on PoW, the DAA increases difficulty exponentially every 10^5 blocks [103, 104]. This is irrelevant in our setting and thus omitted.

Tie Breaking. In Ethereum, a block can have just a single parent. There might be cases where multiple blocks can serve this role, for example if a miner sees multiple blocks at the same height. In such a scenario, it can only choose one of those blocks as a parent for its currently mined block, while the other blocks will be picked as uncles. We will term such events *ties*.

As mining difficulty determines the amount of work invested in each block, Ethereum relies on the notion of the total difficulty (TD) of a block to break ties. The TD of a block is defined as the simple sum of the difficulties of the block itself and all of its main-chain ancestors [34]. In case of ties between blocks, the one with a higher TD is chosen as the parent, and if blocks have equal TDs miners will prefer the one they received earlier [30, 33]. Ethereum's mechanism is an approximation of the one defined by [78] and is an attempt at incentivize publishing newly mined blocks quickly, thus discouraging withholding them [15].

Mining Rewards. We denote the reward received for mining a main-chain block as R ETH. The miner of a block which references an uncle receives $\frac{1}{32}R$. The miner of the referenced uncle receives a reward too, according to the depth of the most recent common shared ancestor of the two: if it is two blocks deep, the uncle's miner gets $\frac{7}{8}R$. The reward diminishes by $\frac{1}{8}R$ for each increase in depth, until reaching 0 ETH [35, 103, 125].

Example 1 provides an example of a block-DAG which is structured according to the aforementioned consensus rules.

Example 1. *We will now go over four blocks and use them to illustrate the various definitions given in Section 4.1. These blocks and their relations are depicted visually in Fig. 1.*

Let block b_0 be the parent of blocks b_1 and u_1, and let block b_2 point to b_1 as its parent, and u_1 as its uncle. Let b_0's timestamp be 0, b_1's timestamp be equal to 26, u_1's timestamp be 27, and b_2 be 32.

If b_0 has a difficulty of 4096, then according to Eq. (1) the difficulties of b_1 and u_1 should be 4094 and 4092, correspondingly. Consequently, according to the same equation b_2's difficulty is 4096.

The standard block-reward for a main-chain Ethereum block which did not refer to uncles is 2 ETH [125], so the miners of b_0 and b_1 earned 2 ETH each, while the miner of b_2 earned 2 ETH, plus $\frac{1}{32} \cdot 2$ ETH

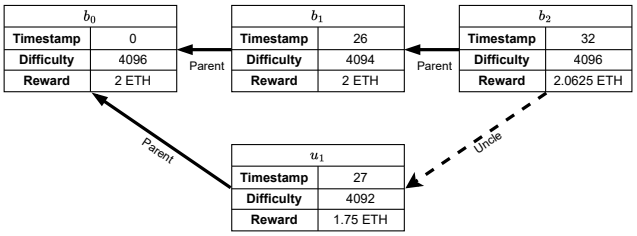

Fig. 1. A graphical depiction of the blocks of Example 1.

for referencing u_1 as its uncle, making the total reward earned by the b_2's miner 2.065 *ETH. Because u_1 is parallel to b_2's parent, the reward earned by u_1's miner is $\frac{7}{8} \cdot 2$ ETH, which amounts to* 1.75 *ETH.*

For an example of real Ethereum blocks which share the same relations as the blocks described here, see Example 2.

4.2 Threat Model

We will now describe the actors that interact with the cryptocurrency and the range of actions available for each one.

Actors. We model the network as being comprised of two parties, an *honest* miner who has a total hash-rate of H hashes-per-second, and an *attacker* that has a hash-rate of A hashes-per-second. We will call the attacker's fraction of the total hash-rate as its *hash-ratio*, and define it as follows: $\alpha \stackrel{\text{def}}{=} \frac{A}{A+H}$.

Note that the honest miner need not necessarily be a single entity, but can be a mining pool or even a number of mining pools, but for the sake of brevity we will mostly refer to it as a single miner. Similarly, the attacker can be a mining pool, or a coalition of mining pools.

Our attack and its analysis do not require looking at more than two consecutive blocks, which according to historical data lasts an average of roughly 13 seconds (see Appendix F). Thus, we follow the literature by assuming that miners cannot obtain or lose hash-power, no new miners enter the network, mining hardware is bought by actors in advance, and all associated costs (such as electricity) are prepaid [50, 59, 67, 102, 110, 132–134]. Indeed, historical data shows that the total hash-rate active on the network does not change by much over such short periods [9].

Attacker Objective. Ethereum, similarly to other cryptocurrencies, has a notion of *fairness* which stipulates that in expectation a miner with a γ fraction of the hash-rate should mine γ of the blocks and obtain a γ fraction of all mining rewards [50, 92, 125].

Our attacker's objective will be to mine more than its fair share of the blocks, e.g. more than α of the blocks. As we will show, this will increase its rewards to be above the fair amount, too.

Honest Mining. We define the *honest* mining protocol similarly to other blockchain papers [50, 59, 110, 127, 128]: our honest miner follows the rules laid down in Ethereum's yellow paper [125]. Thus, the honest miner will always mine on top of the block with highest TD, and will not withhold blocks or manipulate block timestamps.

Allowed Attacker Deviations. Our attacker can rationally deviate from the honest protocol, as long as blocks it produces are indeed valid according to Ethereum's consensus rules [35, 125]. Specifically, the attacker is free to manipulate the timestamps of blocks that it mines to be within the valid range defined in Section 4.1, and can mine on top whichever block it chooses. In order to

create a clean separation between our attack and the previous literature [3, 12, 50, 59, 102, 134], we limit our attacker by forbidding it from withholding blocks.

5 THE UNCLE MAKER ATTACK

We will now describe our attack. Conceptually, the attack manipulates Ethereum's DAA to create blocks which retroactively replace existing main-chain blocks, thus increasing an attacker's share of main-chain blocks beyond its fair share. Honest blocks that were "sidelined" in this manner can later be referred to by main-chain blocks as uncles, thus we will term blocks created by our attacker as *Uncle Makers*.

A major novelty of our attack is that it replaces blocks *without* using block withholding, which is the common method used in the literature [50, 59, 67, 132–134].

Recall that in case of ties between different chains, Ethereum's protocol dictates that the one with a higher TD is picked as the main chain. As Ethereum's DAA assigns mining difficulty according to a block's timestamp, an attacker can set the timestamp for the currently mined block to be low enough to win in case of ties, whether as a preemptive defense mechanism, or to actively replace existing main-chain blocks. Unfortunately, as the block has a higher difficulty, it is harder to mine. We will soon show that this can be avoided.

5.1 Riskless Uncle Making

By carefully picking when to execute the attack, we can make sure that the attacker's blocks will have a higher difficulty when compared to the current tip of the blockchain, and thus will replace it, but still will not have a higher difficulty compared to the block which the attacker would've mined had it been following the honest protocol.

By doing so, we will allow our attacker to execute the attack without incurring any additional mining "risk" when compared to mining honestly: the probability that mining the attack block will succeed will be at least the probability of the attacker successfully mining an honest block, while the attacker's share of the rewards will be higher than its fair portion. We call this attack a *RUM* attack.

Let the tip of the blockchain be block b_1 with a t_1 time difference relative to its parent b_0, and denote the number of seconds that have passed since b_1's timestamp by t_H. According to the honest protocol, the honest miner mines a block, which we will denote by b_H, and sets the block's parent to be b_1, and its timestamp difference to be t_H. See Fig. 2 for a graphical depiction of the aforementioned blockchain state.

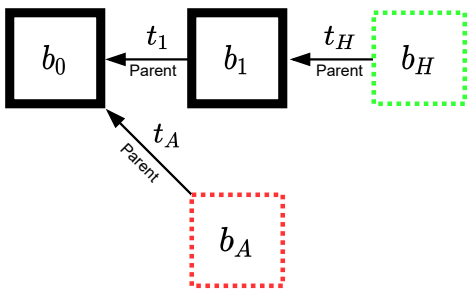

Fig. 2. The blockchain's state when executing a RUM attack.

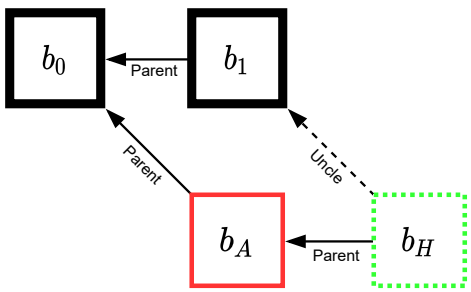

Fig. 3. The blockchain's state after a successful RUM attack.

If $t_1 \in [9, 18)$, then as long as the honest miners still haven't successfully mined a block and $t_H < 9$, an attacker who wishes to execute the RUM attack should mine a block on top of b_0, and set the block's time difference to be some value strictly lower than 9 seconds, for example $t_A \stackrel{\text{def}}{=} 8$ seconds. If the attacker succeeds in mining this block in time, it should be published immediately. In any other case, the attacker should mine according to the honest protocol.

Observe that the time we picked for t_A increases b_A's difficulty compared to b_1. Due to Ethereum's tie-breaking mechanism, if it is indeed mined successfully, it will take b_1's place as the new tip of the main-chain. Thus, the honest miner will pick b_A as the parent of its currently mined block, and b_1 as its uncle, leading the current blockchain state to be as depicted in Fig. 3.

Note that our attack replaced an honest main-chain block with an attacker block without relying on withholding blocks. Instead, the attack actually requires the attacker to *immediately* publish its blocks, standing in stark contrast to other attacks which rely on withholding blocks.

An algorithmic description of the attack is presented in Algorithm 1. The algorithm follows the event-driven model which was used by other works in the field, such as [50, 52].

6 THEORETICAL ANALYSIS

We will now theoretically analyze the RUM attack using a series of theorems, relegating all proofs to Appendix D. Afterwards, we will analyze the block and reward share of an attacker which executes the attack using a Markov decision process (MDP), similarly to other works in the field [50, 59, 63, 67, 102, 127, 133, 134].

In Theorem 1 we prove that the conditions specified in Section 5 for executing a RUM attack are necessary, and that an attacker's probability of successfully mining the attack block is equal to the that of successfully mining an honest block.

Theorem 1. *Let the current tip of the main-chain be block b_1. Denote its parent as b_0, the parent's difficulty as d_0, the timestamp difference between them as t_1, and the difference between the current time and b_1's timestamp as t_H.*

If the following conditions hold, then a rational attacker can execute a RUM attack:

$$\lfloor \frac{t_H}{9} \rfloor = 0, \ \lfloor \frac{t_1}{9} \rfloor = 1$$

Moreso, if $d_0 \geq 2^{22}$, these are the only values for which a RUM attack is possible.

Remark 1. *Any node active on the network, for example our attacker, can learn the values of t_1 and t_H.*

The former, t_1, is publicly available on the blockchain, as valid blocks must include their timestamps [125]. The latter, t_H, can be obtained by sending a standard Ethereum remote procedure call (RPC) request to the honest miner, as responses contain the local time of the responder.

The proof is detailed in full in Appendix D. Briefly, we break the attack down to its different constituents: the blocks generated by the attacker should be valid according to the system's consensus rules, the attacker should successfully replace honest blocks with the attacker's blocks, and mining these blocks should be *riskless*, meaning that it should not be harder when compared to mining honestly. Each constituent is handled via a series of claims which are then combined, culminating with Theorem 1.

The proof of Theorem 1 shows that if the last block's difficulty is lower than 2^{22}, an attacker might have additional timeframes within which an attack is feasible. Remark 2 elaborates on this.

Remark 2. *Although a difficulty of 2^{22} is permitted by Ethereum's DAA, which places the lower difficulty limit at 2^{17}, no block has ever had such a difficulty, the closest being block number 6, which has a difficulty of 2^{32}.*

We proceed to show in Theorem 2 that an attacker's relative share of mainchain blocks exceeds the fair share that can be obtained honestly.

Theorem 2. *Let there be some block b_0. If the attacker uses the RUM mining strategy, its expected relative share of main-chain blocks will be larger than mining honestly, while the absolute number will remain the same.*

As before, the proof is given in Appendix D. The proof relies on analyzing a MDP of the attack, which we describe in Appendix D.3.

In Theorem 3 we prove the corresponding claim regarding an attacker's share of the rewards.

Theorem 3. *For any block b_0, an attacker can increase its expected absolute and relative rewards by using the RUM mining strategy instead of mining honestly.*

The proof for Theorem 3 is similar to that of Theorem 2. Again, it is given in full in Appendix D.

Corollary 1. *As Theorems 2 and 3 show, both the relative and absolute share of blocks and rewards obtained by the attacker are higher when compared to mining honestly, thus a direct consequence is that the relative and absolute share of honest miners are* lower*, meaning that the attack not only benefits the attacker, but also harms its competitors.*

As Theorem 1 shows, the attack is riskless, meaning that the probability of successfully mining an attack block is equal to that of mining an honest block. By combining all of these results, one can obtain that the RUM mining strategy dominates the honest one.

7 IN SEARCH OF LOST TIME: UNCLE MAKING IN THE WILD

We will now attempt to give a conclusive answer to a long-standing question: *do miners attack the consensus layer of major cryptocurrencies?* The surprising answer is a resounding **yes!**

To reach this affirmative answer, we crawled Ethereum's blockchain using a standard Ethereum full node and collected relevant data for all main-chain and uncle blocks starting from block 15, 226, 042, down to 12, 000, 000. All the data we collected, and the code used to collect this data and generate all the graphs presented in this section are provided in Appendix F.

Most previous works usually only attempted to find evidence for block withholding attacks, and have tried doing so by planting many nodes throughout the network, collecting data and then analyzing it using various statistical methods. On the other hand, we show that the evidence proving that miners attack the consensus layer is hiding in plain sight and can be obtained using a *single* node and closely inspecting publicly available on-chain data. We cover these related works and provide a more in-depth comparison with the current paper in Section 9.

7.1 Ethereum Mining Pools

Using the previously mentioned data, we identified Ethermine as the largest mining pool currently active in Ethereum, consistently mining at least 22% of all main-chain blocks, more than any other mining pool in the past 3 years.

According to our analysis, the second largest mining pool for the same time-frame is F2Pool, which roughly amounts to about half as many main-chain blocks as Ethermine. Notably, F2Pool has a considerable amount of hash-rate active on other cryptocurrencies, such as Bitcoin [82]. The pool's founder has made a relatively well publicized condemnation of competing mining pools, blaming them for attacking his own mining pool, but without providing any concrete proof [115]. This is in stark contrast to the evidence that we uncover in this work which shows that, in fact, *F2Pool are attacking other mining pools.*

7.2 Identifying Uncle Maker Attacks

Before we examine real-world data, we would like to first crystallize a few insights which will allow us to identify uncle maker-esque attacks.

Miners that execute uncle maker attacks should have a higher than expected main-chain block share and lower than expected uncle block share. Recall that the RUM attack allows an attacker to replace main-chain blocks mined by other miners with a block of its own. Thus, it follows that miners which perform the attack will have fewer uncles and more main-chain blocks when compared to honest miners.

Uncle maker attacks can be observed from block timestamps. As our attack relies on manipulating block timestamps, one can attempt to uncover blocks which were created as part of such an attack by carefully analyzing the timestamps of historical Ethereum blocks.

By definition, the RUM attack requires an attacker to falsely set block timestamps to be earlier than they are. Thus, it is reasonable to expect that main-chain blocks created by an attacker will have an over-representation of low timestamp differences relative to their parents.

In comparison, timestamp differences between honest blocks and their parents should be distributed according to the exponential distribution [8, 13, 49, 50, 106, 109], with a slight under-representation of low timestamp differences due to propagation delay in the network [57, 72].

Miners usually use a publicly identifiable coinbase address. In order for a miner to receive the rewards for the blocks that it mines, the miner must include an address to transfer the reward to within a mined block, also called a *coinbase* address. Large mining pools commonly use a specific address for their block-rewards and advertise it, in the hopes it might attract new participants or help assert "political" control over the mined cryptocurrency [57].

In spite of this, there is no mechanism in place to force mining pools to stick to a single address throughout their lifespan, and pools are free to change their address to a new secret one at will. For example, mining pools might do so when executing an attack, in order to cover their tracks.

Still, it is common for works in the field to assume that mining pools do identify themselves in a truthful manner [57, 72, 91, 107, 120, 130]. Indeed, a large percentage of all Ethereum blocks belong to miners who identify their addresses using sites such as [44, 88].

7.3 Block Shares

We will now attempt to apply our previous insights to real-world Ethereum data, starting with the main-chain and uncle shares of the 4 largest mining pools in Ethereum, given in Fig. 4 and Fig. 5, respectively. Note that these top 4 pools have mined a combined share of more than 50% of both main-chain and uncle blocks.

Previous works have shown that larger pools such as Ethermine have a "size-advantage" leading them to win in cases of ties more often when compared to smaller pool like F2Pool. This leads them to having a higher share of main-chain blocks and a lower share of uncle blocks when compared to smaller mining pools, solely due to having more computational resources at their behest [79, 82].

By comparing Figs. 4 and 5, it is apparent that F2Pool has considerably fewer uncles than is expected for a mining pool of their size: Ethermine and $0x5a \ldots 4c$ maintain their respective positions with regards to both mainchain blocks and uncles, with Ethermine having the largest share of both, and $0x5a \ldots 4c$ having the third largest share. On the other hand, F2Pool has the second largest share of mainchain blocks, but drops to the fourth place when it comes to uncle blocks.

A metric which is commonly used to compare mining pools is the *uncle rate*, defined per-miner as the ratio between the number of uncles which the miner has mined for some time period, and the

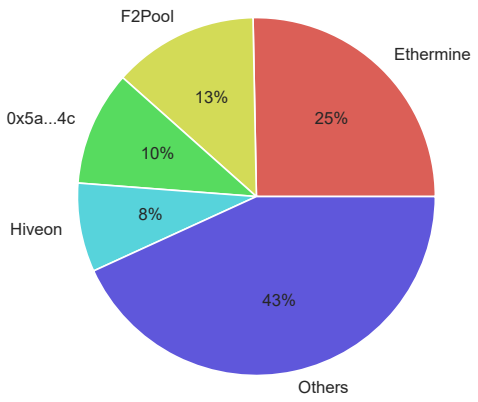

Fig. 4. The 4 largest mining pools' share of main-chain blocks.

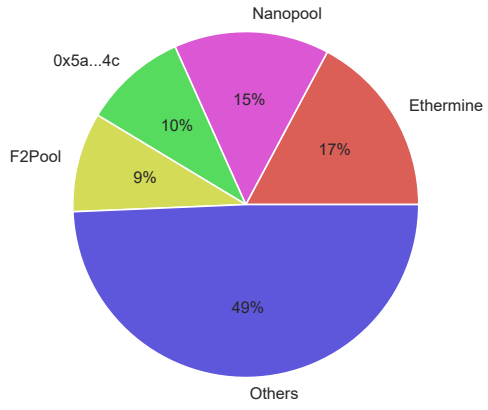

Fig. 5. The 4 largest mining pools' share of uncle blocks.

total amount of blocks of all kinds (uncle and main-chain) it mined during the same period. A lower rate is better due to main-chain blocks receiving a higher block-reward and earning the fees for all transactions contained within them. Fig. 6 depicts this metric over time for both Ethermine and F2Pool. Although both were relatively comparable in the year between July 2019 and September 2020, F2Pool took the lead starting in October 2020.

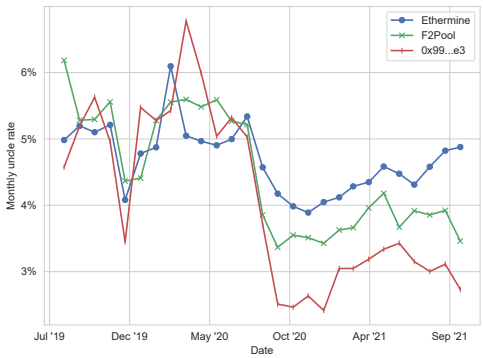

Fig. 6. The monthly uncle rate for Ethermine, F2Pool and $0x99\ldots e3$, defined per-miner as the ratio between the number of uncle blocks it mined in some month and the total amount of blocks it mined that month.

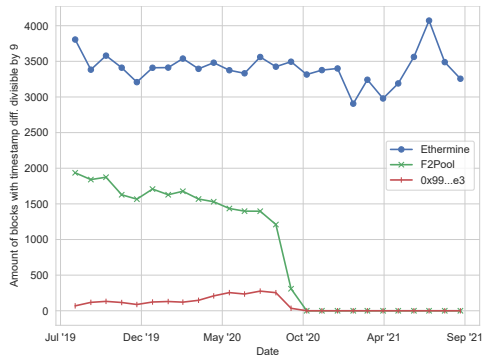

Fig. 7. The monthly amount of main-chain blocks with a timestamp difference from their parent which is divisible by 9. Note that both F2Pool and $0x99\ldots e3$ stop mining such blocks after October '20.

One could argue that F2Pool might have achieved this feat by investing in good network connectivity to other network nodes, thus propagating its blocks faster, leading its blocks to win ties by virtue of arriving to peers earlier. Indeed, at the end of October 2020, roughly when F2Pool's uncle rate has started improving, a company called bloXroute has published a blog-post describing how it helped reduce F2Pool's uncle rate by improving it's networking layer [18].

In order to rule out this possibility, we will now turn to analyzing the timestamps of blocks over this period.

7.4 Block Timestamps

Recall that according to Ethereum's documentation [31, 54, 125], the target time-difference between the mining of two blocks is 12 – 14 seconds. Indeed, the average time difference between mainchain blocks and their parents is 13.45 seconds, with the same comparison between uncle blocks and their parents yielding another reasonable average of 13.81 seconds.

The average difference might obscure more minute details, leading us to plot a histogram of the timestamp differences between main-chain blocks and their parents in Fig. 8, and between uncles and their parents in Fig. 9. Both histograms were truncated at 18 seconds to maintain readability.

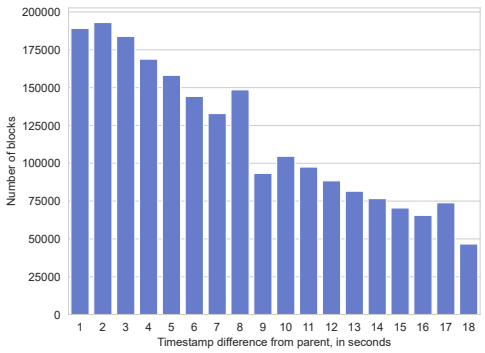

Fig. 8. Total number of main-chain blocks with a given timestamp difference from their parent, since block 12, 000, 000.

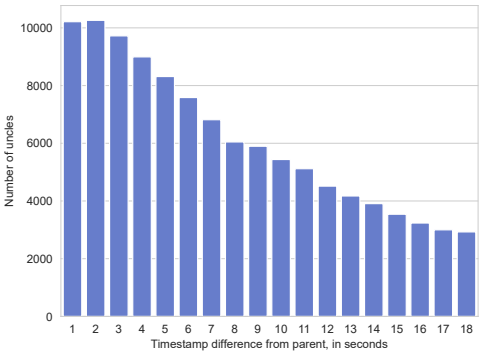

Fig. 9. Total number of uncle blocks with a given timestamp difference from their parent, since block 12, 000, 000.

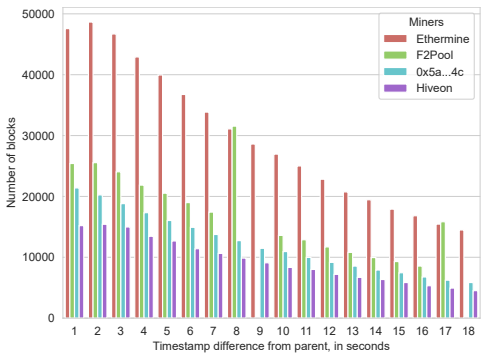

Fig. 10. Number of main-chain blocks with a given timestamp difference from their parent, separated by mining pool, for the 4 largest mining pools. Note that F2Pool has no blocks with timestamp differences divisible by 9, and an over-representation for a difference of 8 seconds.

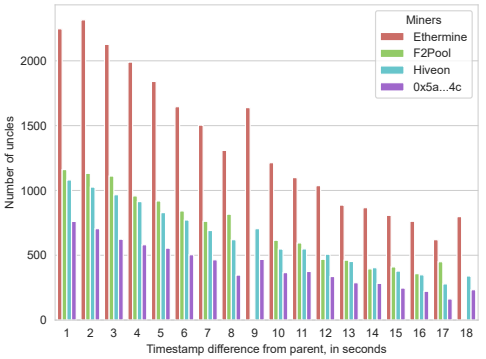

Fig. 11. Number of uncle blocks with a given timestamp difference from their parent, separated by mining pool, for the 4 largest mining pools. Note that pools except F2Pool have an over-representation of blocks with timestamp differences divisible by 9, due to F2Pool's attack.

Fig. 9 looks precisely as predicted in Section 7.2 with regards to the honest scenario: it has a slight under-representation at the 1 second mark, and its trend seems to fit the expected exponential distribution.

On the other hand, Fig. 8 has quite a sizable under-representation of timestamp differences which are divisible by 9, and an over-representation at seconds 8 and 17. This seems to match the expected fingerprint of an uncle maker-style attack.

7.5 Catching F2Pool Red-handed

In order to get to the bottom of the anomalies we observed so far, we created more detailed histograms which separate blocks according to the identity of their miner. Specifically, in Fig. 10 we plot the per-miner histogram for the timestamp difference between main-chain blocks and their parents, and in Fig. 11 we do the same for uncles.

Figs. 10 and 11 explain the reason for F2Pool's advantage very clearly: F2Pool did not mine even a single main-chain block or uncle with a timestamp difference which is divisible by 9 since block 11, 064, 754, which was mined almost two years ago. Whenever the honest timestamp difference should have been 9 seconds, they instead have falsely set it to be 8, as required to execute an uncle maker attack. We provide an example of a suspected uncle maker block by F2Pool in Example 2.

Fig. 11 shows that F2Pool's foolhardy behavior harms other mining pools, as can be deduced by looking at the larger-than-expected number of uncles with a timestamp difference of 9 seconds mined by Ethermine, Hiveon and an unnamed pool which uses the address $0x5a \ldots 4c$. This is due to F2Pool using a timestamp manipulation tactic which precisely targets this time difference window, while they mine honestly at all other windows.

7.6 F2Pool's Attack is not Riskless

We proved in Section 6 that in order for an uncle maker attack to be riskless, very specific conditions must hold. These conditions preclude the possibility of executing a riskless attack in certain time windows, for example if the time since the last valid block was observed is at least 18 seconds. Yet, as Fig. 12 shows, F2Pool do not shy from executing uncle maker-style attacks even at later periods but only for time differences which are divisible by 9, leading us to believe they employ a variant of the attack.

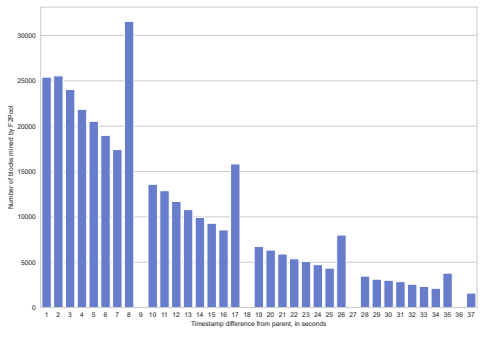

Fig. 12. Number of main-chain blocks mined solely by the F2Pool mining pool, with a given timestamp difference from their parent, since block 12, 000, 000.

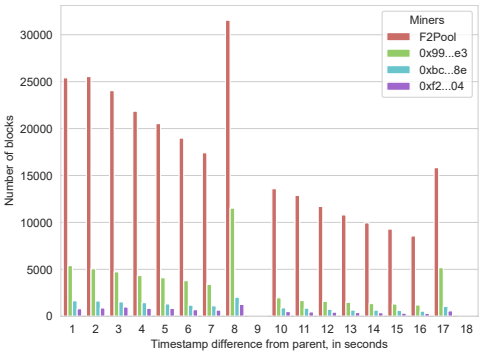

Fig. 13. Number of main-chain with a given timestamp difference from their parent, separated by mining pool, for mining pools suspected to manipulate timestamps.

7.6.1 Risky Uncle Maker Attack. One possibility is that F2Pool executes the attack if the *expected* profit from it is higher when compared to mining honestly, *even if it incurs additional risk*. E.g. even if the probability of mining the attack block is lower than the probability of mining an honest one. Due to this, we call this variant the *risky uncle maker* attack.

Such a risky uncle maker attack can be profitable in expectation under certain scenarios, for example if another miner "snatched" transactions which pay exuberantly high fees and included them in its block before the attacker, or if the attacker has a sufficiently high hash-rate and good network connectivity.

7.6.2 Preemptive Uncle Maker Attack. Another option is that F2Pool, instead of actively trying to replace existing blocks, actually attempts to preemptively assure their blocks will win in case of ties by setting block timestamps to be within an earlier 9 second window when compared to the honest timestamps. This increases mining difficulty and makes sure their blocks have a higher TD, meaning that their blocks will be picked as main-chain blocks over competing honest blocks. We term this variant of the attack as a *preemptive uncle maker* attack.

This type of attacks allows miners not to waste computation resources by mining on top of a tip that has been made "stale" by another block mined on top of it, but which has not been received yet. If that is the case, attackers setting their clock to a second earlier can be assured that their mined block will replace any block honestly mined in specific time intervals, which might be disseminated through the network currently, but hasn't reached the attackers yet.

7.6.3 RUM is Hard to Implement. Lastly, we would like to raise the possibility that F2Pool chose to implement the attack which we observe in the wild simply because it was easier to implement than any other attack, including RUM. In fact, we implemented an arguably less outrageously apparent variant of the observed attack as a patch for geth. In this patch we set geth's clock to be late by exactly one second. This achieves the same effect as F2Pool's attack, while producing a more natural-looking timestamp difference curve. We provide additional details in Appendix F.

7.7 Other Attackers

Although F2Pool is the largest mining pool which is currently engaging in uncle maker-esque attacks, it is not the only one; Fig. 13 shows the timestamp histogram of three additional miners which use the same exact attack, meaning that they abstain from using timestamp differences which are divisible by 9.

The second largest such attacker after F2Pool is $0x99\ldots e3$. This miner did not have a name in Etherscan's database, and thus we will call him *Sneezy*. Sneezy's histogram spikes at the 8 seconds mark to be almost twice as high than the corresponding bin at the 1 second mark, while F2Pool's 8 second bin is only roughly 20% taller, suggesting that Sneezy executes the attack over longer periods of time.

Indeed, as Fig. 6 shows, this aggressive attack is worthwhile, and has decreased Sneezy's uncle rate considerably, even moreso than it has benefited F2Pool. Fig. 7 shows the monthly amount of blocks with a timestamp difference which is divisible by 9 for each one of Ethermine, F2Pool and Sneezy. While Ethermine's graph looks relatively uneventful, F2Pool's and Sneezy's drop to 0 at almost the same time. A closer inspection of the blockchain reveals that Sneezy started the attack at block $11,080,310$, meaning slightly more than two days after F2Pool.

8 MITIGATION

We will now go over mitigation techniques that arise from the previously discussed results. It is well known that cryptocurrencies are leary of making changes to their consensus mechanisms, oftentimes *forking* the blockchain and thus creating two communities, one which abides to the

previous consensus rules and rejects the change, and another which adopts it [71, 74]. Thus, we would like to focus on solutions which are reasonable in scope, e.g. similar to previously adopted Ethereum improvement proposals (EIPs) [37].

8.1 Minimize attacker flexibility by increasing the minimal difficulty

In the proofs given in Section 6, we have shown that if the minimal difficulty is strictly less than 2048^2, then an attacker could execute a riskless attack in less restrictive conditions than those described in Theorem 1.

As we have mentioned in the same section, mining difficulty did not come close to such a difficulty in the seven years since Ethereum was launched, but the fix is worthwhile for any small cryptocurrency which is based on Ethereum's mechanism and might reach such low levels of mining difficulty.

The above mitigation was implemented in Appendix F.7.

8.2 Reject competing chains more aggressively

The RUM attack and the variants which we observe in the wild can be mitigated completely by rejecting competing chains in a more aggressive manner.

More concretely, denote the tip of the competing chain by b' and of the local chain by b. A miner should reject it if all of the following conditions hold:

(1) If b and b' share a parent
(2) If b has at least the same number of uncles as b'
(3) If the timestamp of b' is less than the miner's local clock by one second or more

If at least one of these conditions do not hold, the miner should adhere to Ethereum's existing protocols.

Notice that an attack block for either RUM or the variant which we observe in the wild must fulfill all of these conditions, thus if all honest miners would reject competing chains based on this criteria, the attack will be prevented.

We would like to mention that the third condition is actually not as strict as it appears. According to [72], 85% of blocks are propagated in less than 100 milliseconds, while in recent history propagation time did not exceed 650 milliseconds [6, 10, 48].

Additionally, although this mitigation is similar to already existing Ethereum consensus rules which use a node's local clock to reject blocks in some cases, we emphasize that such considerations might open the door for various attacks. Specifically, we have not analyzed the implications of this mitigation with regards to any other attack besides the RUM attacks and the variations which we have covered in this work.

We have implemented this mitigation in Appendix F.7.

8.3 Migrate to Other Mechanisms

An obvious mitigation technique which will solve both this attack, and any other PoW-related one, is to migrate Ethereum's consensus mechanism to proof-of-stake (PoS). This transition, which is also called *The Merge* [39], is scheduled to happen by the end of 2022 (though it has been scheduled to happen since 2017 and has been postponed multiple times by now [16]). Alternatively, it is possible to other protocols with theoretical guarantees, like [56, 92], and even [109], which Ethereum's current mechanism is based on.

Other solutions which might be smaller in scope and thus easier to implement are to adopt better fork-choosing rules [131, 132], use reliable timestamps [1, 4, 28, 75] or avoid using timestamps for difficulty adjustments altogether [108].

9 RELATED WORK

We will now review the major related works of the field and compare these to our paper, when appropriate. A summary is provided in Table 1. Additionally, in order to paint a complete picture of the current research landscape, we surveyed a few additional related papers in Appendix E.

Withholding Attacks. Most consensus-level attacks rely on block withholding and on an attacker's ability to both immediately hear about new blocks, and broadcast withheld blocks to some fraction of the network before this fraction hears of these new blocks. Usually, such attacks theoretically allow miners to increase *expected* profits, although they do have a non-zero probability of failing and causing a loss. As far as we know, there is currently no software in place to facilitate this attack.

In contrast, we do not require such assumptions, instead relying on an attacker's ability to manipulate timestamps. This is indeed feasible, as seen in our empirical evaluation and by the geth patch we provide which implements the attack.

Examples of such attacks include the celebrated *Selfish Mining* attack [50], which increases an attacker's share of the blocks beyond its share of the mining power. This is achieved by mining blocks and withholding them. Withheld blocks are published in a manner which discards blocks mined by honest parties. Variants of the attack were further explored in Bitcoin by [23, 59, 102] and in Ethereum by [52, 61, 62, 98]. A notable extension of selfish mining is the *Stubborn Mining* attack, in which attackers continue mining on selfish forks even if these trail behind the honest chain [81, 91, 123].

As both selfish and stubborn mining are not proven to be optimal mining strategies, several works have attempted creating general frameworks for mining attacks which rely on block withholding, such as [67, 132–134], which employ machine learning (ML) techniques to recreate classic results and derive new ones. These have the same pitfalls as the selfish-mining papers.

Manipulating DAAs Without Withholding Blocks or Manipulating Timestamps. A strand of research focused on directly manipulating DAAs to increase mining profits, thus avoiding block withholding. All such attacks are not currently known to be implemented in software. In contrast, our attack can be executed by applying a simple software patch.

For example, in *Coin Hopping* attacks miners "hop" between cryptocurrencies and mine the one which is currently the most profitable [86]. Certain DAAs are susceptible to cyclical mining difficulties, which can be exploited by miners that join when the difficulty is low and leave when it is high [124]. These attacks assume miners can indeed "hop" between mining different cryptocurrencies in a minute's notice and fixed exchange-rates between the attacked cryptocurrencies, but these are known to be extremely volatile and do not always move in unison [68, 128].

Similarly but in the scope of just a single cryptocurrency, *Energy Equilibria* attacks show that Bitcoin miners can both minimize operational costs and manipulate mining difficulty to increase the average amount of rewards per unit of time by repeatedly turning their mining hardware on and off, but the profitability of this mining strategy for a specific miner depends on the fraction of mining expenses which it dedicates to electricity and whether this electricity is bought in advance or not [53, 60].

On the other hand, the *Stretch & Squeeze* attack presented in [127] focuses on increasing and decreasing Bitcoin's and Ethereum's block-rates, and then using this to create interest-rate arbitrage between decentralized finance (DeFi) lending platforms, which can be exploited to make a profit by taking a large collateralized loan from one platform and depositing it in another. The market's response to such an attack has not been explored, and it is reasonable to assume that such arbitrage will be closed by other players, thus voiding potential gains from being made.

Timestamp Attacks. In Ethereum, timestamps have been mostly used to attack smart contracts [17]. An exception is the *Stretch & Squeeze* paper [127], that found two timestamp weaknesses in geth's code. The first is a simple bug that was quickly fixed. The second weakness is inherent to Ethereum's consensus and still remains; it allows miners to intentionally mine uncles with a difficulty which is lower by 5% compared to the honest one. This can be used to increase mining difficulty and slow-down the blockrate, but was not included in the analysis performed in the paper.

The *Difficulty Raising* attack presented by [5] is an attack on Bitcoin's DAA. Similarly to our attack, it relies on manipulating timestamps in order to increase mining difficulty and replace existing blocks. In contrast, it focuses on Bitcoin, and as claimed by the work, the attack is infeasible and entirely theoretical. A framework for Bitcoin that allows deriving conditions which eliminate the theoretical possibility of such an attack is analyzed in [56].

Real-world Evidence of Attacks. Many have wondered why selfish mining and related attacks are not observed in the wild [67, 91], with one glaring reason being the lack of concrete software implementations to facilitate such attacks [117]. Some previous works relied on statistical methods to detect abnormalities such as a large amount of blocks consecutively mined by a single miner, or a high number of forks, but have not found any conclusive evidence of a major attack [79, 80], while [82] found that one Bitcoin mining pool performs coin-hopping, specifically between Bitcoin and Bitcoin Cash, the latter of which is a fork of Bitcoin [74].

Two exceptions are [82, 127]. The former used data from Ethereum nodes to compare block timestamps with the actual block arrival times, and has concluded that the two are roughly similar. The latter claims that the last three years of Ethereum blocks do not bear the mark of timestamp manipulations, and has done so by looking at basic metrics such as the mean and median differences between blocks and their direct ancestors, which indeed are not indicative of any apparent manipulation. As we've shown, by looking at the histogram of timestamp differences, it becomes clear that, in fact, miners disregard the honest protocol and set timestamps to gain an unfair advantage.

10 CONCLUSIONS

In this work, we have presented a novel attack on Ethereum's consensus mechanism and multiple variations on it, including an implementation of one such variation as a patch for geth, Ethereum's most popular client.

We have analyzed this attack and have proved that miners can execute it in a risk-less manner, thereby increasing both their relative share of blocks and their absolute and relative share of block rewards.

We have shown that this attack has been executed by Ethereum's second largest mining pool, F2Pool, for over two years, thereby finding the first-ever proof of an attack on the consensus mechanism of a major cryptocurrency.

Finally, we have suggested concrete mitigation techniques which can be quickly implemented until Ethereum's migration to PoS is finalized.

AVAILABILITY

To maintain anonymity, all code and data used in this paper were uploaded to a Google Drive account which was opened under the pseudonym *Uncle Maker*. This account can be accessed using the following link.

Details about our data sources, installation and usage instructions are provided in Appendix F.

ACKNOWLEDGMENTS

The authors are partially supported by grants from the Ministry of Science & Technology, Israel, the Israel Science Foundation (grants 1504/17 & 1443/21) and by a grant from the Hebrew University of Jerusalem, Israel (HUJI) Federmann Cyber Security Research Center in conjunction with the Israel National Cyber Directorate.

REFERENCES

[1] Aydin Abadi, Michele Ciampi, Aggelos Kiayias, and Vassilis Zikas. 2020. Timed Signatures and Zero-Knowledge Proofs—Timestamping in the Blockchain Era—. In *International Conference on Applied Cryptography and Network Security* (2020). Springer, Springer International Publishing, 335–354. https://doi.org/10.1007/978-3-030-57808-4_17

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*, Matteo Maffei and Mark Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–186.

[3] Sarah Azouvi and Alexander Hicks. 2020. SoK: Tools for Game Theoretic Models of Security for Cryptocurrencies. https://doi.org/10.21428/58320208.8e7f4fab arXiv:1905.08595 [cs.CR]

[4] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2019. Ouroboros Chronos: Permissionless Clock Synchronization via Proof-of-Stake. *IACR Cryptol. ePrint Arch.* 2019 (2019), 838. https://eprint.iacr.org/2019/838

[5] Lear Bahack. 2013. Theoretical Bitcoin Attacks with less than Half of the Computational Power (draft). https://doi.org/10.48550/ARXIV.1312.7013

[6] BDN. 2022. Blockchain Distribution Network. https://www.bdn-explorer.com/

[7] Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shinichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner. 2021. SoK: Algorithmic Incentive Manipulation Attacks on Permissionless PoW Cryptocurrencies. In *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12676)*. Springer, 507–532. https://doi.org/10.1007/978-3-662-63958-0

[8] George Bissias and Brian N Levine. 2020. Bobtail: Improved Blockchain Security with Low-Variance Mining. In *ISOC Network and Distributed System Security Symposium* (2020). Internet Society. https://doi.org/10.14722/ndss.2020.23095

[9] BitInfoCharts. 2022. Bitcoin, Ethereum, Dogecoin, XRP, Ethereum Classic, Litecoin, Monero, Bitcoin Cash, Zcash, Bitcoin Gold Hashrate historical chart. https://web.archive.org/web/20220522122528/https://bitinfocharts.com/comparison/hashrate-btc-eth-doge-xrp-etc-ltc-xmr-bch-zec-btg.html#3y

[10] BLOCKCYPHER. 2022. Blockchain Web Services. https://www.blockcypher.com

[11] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.

[12] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. 2015. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE symposium on security and privacy*. IEEE, IEEE, 104–121. https://doi.org/10.1109/SP.2015.14

[13] R. Bowden, H. P. Keeler, A. E. Krzesinski, and P. G. Taylor. 2018. Block arrivals in the Bitcoin blockchain. *ArXiv e-prints* (Jan. 2018). arXiv:1801.07447 [cs.CR]

[14] Vitalik Buterin. 2016. EIP-100: Change difficulty adjustment to target mean block time including uncles. https://eips.ethereum.org/EIPS/eip-100

[15] Vitalik Buterin. 2022. Ethereum Whitepaper. https://web.archive.org/web/20220728020709/https://ethereum.org/en/whitepaper/

[16] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437* (Oct. 2017). arXiv:1710.09437 [cs.CR]

[17] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2019. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses. https://doi.org/10.1145/3391195 arXiv:1908.04507 [cs.CR]

[18] Shen Chen. 2020. The Secret Weapon F2Pool Used to Tackle Its Uncle Rate. https://web.archive.org/web/20220806080656/https://medium.com/bloxroute/the-secret-weapon-f2pool-used-to-tackle-its-uncle-rate-1ecb6fe47ef8

[19] Ethereum Classic. 2017. Ethereum Classic. *Ethereum Classic (accessed 16 October 2017) https://ethereumclassic. github. io* (2017).

[20] Ethereum Classic. 2022. Proof of Work. https://web.archive.org/web/20220519111931/https://ethereumclassic.org/why-classic/proof-of-work/

[21] Nicolas T Courtois and Lear Bahack. 2014. On subversive miner strategies and block withholding attack in bitcoin digital currency. *arXiv preprint arXiv:1402.1718* (2014).
[22] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 910–927. https://doi.org/10.1109/SP40000.2020.00040
[23] Michael Davidson, Tyler Diamond, et al. 2020. On the Profitability of Selfish Mining Against Multiple Difficulty Adjustment Algorithms. *IACR Cryptol. ePrint Arch.* 2020 (2020), 94.
[24] Monika Di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*. IEEE, 69–78.
[25] Docker. 2022. Get Started with Docker. https://www.docker.com/get-started/
[26] Maya Dotan and Saar Tochner. 2021. Proofs of Useless Work – Positive and Negative Results for Wasteless Mining Systems. arXiv:2007.01046 [cs.CR]
[27] David Easley, Maureen O'Hara, and Soumya Basu. 2019. From mining to markets: The evolution of bitcoin transaction fees. *Journal of Financial Economics* 134, 1 (2019), 91–109.
[28] Gabriel Estevam, Lucas M. Palma, Luan R. Silva, Jean E. Martina, and Martín Vigil. 2021. Accurate and decentralized timestamping using smart contracts on the Ethereum blockchain. *Information Processing & Management* 58, 3 (2021), 102471. https://doi.org/10.1016/j.ipm.2020.102471
[29] Ethereum. 2020. Block timestamps. https://git.io/JLQVd
[30] Ethereum. 2021. forkchoice.go. https://github.com/ethereum/go-ethereum/blob/be9742721f56eb8bb7ebf4f6a03fb01b13a05408/core/forkchoice.go#L43
[31] Ethereum. 2021. Geth's difficulty adjustment code. https://git.io/JXuhA
[32] Ethereum. 2021. protocol_params.go. https://github.com/ethereum/go-ethereum/blob/377c7d799ff62c6060939a4f95532df93a345f63/params/protocol_params.go#L168
[33] Ethereum. 2022. backend.shouldPreserve. https://github.com/ethereum/go-ethereum/blob/ecae8e4f655775bf6935543e3e9136566f4823a2/eth/backend.go#L404
[34] Ethereum. 2022. blockchain.writeBlockWithState. https://github.com/ethereum/go-ethereum/blob/be9742721f56eb8bb7ebf4f6a03fb01b13a05408/core/blockchain.go#L1219
[35] Ethereum. 2022. consensus.go. https://github.com/ethereum/go-ethereum/blob/a52bcccfe1fd9b10d27bff1121a8465982af7714/consensus/ethash/consensus.go#L44
[36] Ethereum. 2022. consensus.makeDifficultyCalculator. https://github.com/ethereum/go-ethereum/blob/377c7d799ff62c6060939a4f95532df93a345f63/consensus/ethash/consensus.go#L404
[37] Ethereum. 2022. Ethereum Improvement Proposals. https://web.archive.org/web/20220710202618/https://eips.ethereum.org/
[38] Ethereum. 2022. Go Ethereum. https://web.archive.org/web/20220512071235/https://geth.ethereum.org/
[39] Ethereum. 2022. The Merge. https://ethereum.org/en/upgrades/merge/
[40] Ethereum. 2022. Nodes and clients. https://github.com/ethereum/ethereum-org-website/blob/35c262525ae69bdcfee7e35ad6bc3046d282fc56/src/content/developers/docs/nodes-and-clients/index.md?plain=1#L1
[41] ethereum/devp2p. 2017. DEV's p2p network protocol & framework. https://web.archive.org/web/20220729125908/https://gitter.im/ethereum/devp2p?at=59bea341614889d4750c29ac
[42] EthereumPoW. 2022. EthereumPoW. https://web.archive.org/web/20220806223601/https://ethereumpow.org/
[43] Ethernodes. 2021. The popularity of Ethereum clients. ethernodes.org.
[44] Etherscan. 2021. Top 25 Miners by Blocks. https://web.archive.org/web/20210930170721/https://etherscan.io/stat/miner/
[45] Etherscan. 2022. Blocks. https://etherscan.io/blocks
[46] Etherscan. 2022. Ethereum Network Transaction Fee Chart. https://etherscan.io/chart/transactionfee
[47] Etherscan. 2022. Uncles. https://etherscan.io/uncles
[48] ethstats. 2022. Ethereum Network Status. https://ethstats.net/
[49] Ittay Eyal. 2015. The Miner's Dilemma. In *2015 IEEE Symposium on Security and Privacy*. 89–103. https://doi.org/10.1109/SP.2015.13
[50] Ittay Eyal and Emin Gün Sirer. 2014. Majority is not enough: Bitcoin mining is vulnerable, In International conference on financial cryptography and data security. *Commun. ACM* 61, 436–454. https://doi.org/10.1145/3212998
[51] Eugene A Feinberg and Adam Shwartz. 2012. *Handbook of Markov decision processes: methods and applications*. Vol. 40. Springer Science & Business Media.
[52] Chen Feng and Jianyu Niu. 2019. Selfish Mining in Ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1306–1316. https://doi.org/10.1109/ICDCS.2019.00131

[53] Amos Fiat, Anna Karlin, Elias Koutsoupias, and Christos Papadimitriou. 2019. Energy Equilibria in Proof-of-Work Mining. In *Proceedings of the 2019 ACM Conference on Economics and Computation* (Phoenix, AZ, USA) *(EC '19)*. Association for Computing Machinery, New York, NY, USA, 489–502. https://doi.org/10.1145/3328526.3329630

[54] Ethereum Foundation. 2022. Blocks. https://github.com/ethereum/ethereum-org-website/blob/e98c23119c1514f46b2bcdcc8b2ea59154069bbd/src/content/developers/docs/blocks/index.md?plain=1#L57

[55] Mark Friedenbach. 2018. Forward Blocks.

[56] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2017. The bitcoin backbone protocol with chains of variable difficulty. In *Annual International Cryptology Conference*. Springer, 291–323.

[57] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. 2018. Decentralization in Bitcoin and Ethereum Networks. In *Financial Cryptography and Data Security*, Sarah Meiklejohn and Kazue Sako (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–457.

[58] A. Gervais, G. O. Karame, V. Capkun, and S. Capkun. 2014. Is Bitcoin a Decentralized Currency? *IEEE Security Privacy* 12, 3 (May 2014), 54–60. https://doi.org/10.1109/MSP.2014.49

[59] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/2976749.2978341

[60] Guy Goren and Alexander Spiegelman. 2019. Mind the mining. In *Proceedings of the 2019 ACM Conference on Economics and Computation*. ACM, 475–487. https://doi.org/10.1145/3328526.3329566 arXiv:1902.03899

[61] Cyril Grunspan and Ricardo Pérez-Marco. 2019. Selfish Mining and Dyck Words in Bitcoin and Ethereum Networks. https://doi.org/10.48550/ARXIV.1904.07675

[62] Cyril Grunspan and Ricardo Pérez-Marco. 2019. Selfish Mining in Ethereum. https://doi.org/10.48550/ARXIV.1904.13330

[63] Runchao Han, Zhimei Sui, Jiangshan Yu, Joseph Liu, and Shiping Chen. 2021. Fact and fiction: Challenging the honest majority assumption of permissionless blockchains. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 817–831.

[64] Timo Hanke. 2016. AsicBoost - A Speedup for Bitcoin Mining. *arXiv e-prints*, Article arXiv:1604.00575 (Apr 2016), arXiv:1604.00575 pages. arXiv:1604.00575 [cs.CR]

[65] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585 (2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[66] Jerry R Hobbs and Feng Pan. 2006. Time ontology in OWL. *W3C working draft* 27, 133 (2006), 3–36.

[67] Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. 2021. SquirRL: Automating Attack Analysis on Blockchain Incentive Mechanisms with Deep Reinforcement Learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://doi.org/10.14722/ndss.2021.24188

[68] Paraskevi Katsiampa, Shaen Corbet, and Brian Lucey. 2019. High frequency volatility co-movements in cryptocurrency markets. *Journal of International Financial Markets, Institutions and Money* 62 (2019), 35–52. https://doi.org/10.1016/j.intfin.2019.05.003

[69] Jonathan Katz and Yehuda Lindell. 2020. *Introduction to modern cryptography*. CRC press. https://doi.org/10.1201/9781351133036

[70] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388. https://doi.org/10.1007/978-3-319-63688-7_12

[71] Lucianna Kiffer, Dave Levin, and Alan Mislove. 2017. Stick a fork in it: Analyzing the Ethereum network partition. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 94–100.

[72] Lucianna Kiffer, Asad Salman, Dave Levin, Alan Mislove, and Cristina Nita-Rotaru. 2021. Under the hood of the ethereum gossip protocol. In *International Conference on Financial Cryptography and Data Security*. Springer, 437–456.

[73] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference 2018*. 91–104.

[74] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, and Yongdae Kim. 2019. Bitcoin vs. Bitcoin cash: Coexistence or downfall of bitcoin cash?. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 935–951.

[75] Esteban Landerreche, Marc Stevens, and Christian Schaffner. 2020. Non-interactive cryptographic timestamping based on verifiable delay functions. In *International Conference on Financial Cryptography and Data Security*. Springer, Springer International Publishing, 541–558. https://doi.org/10.1007/978-3-030-51280-4_29

[76] Ron Lavi, Or Sattath, and Aviv Zohar. 2019. Redesigning Bitcoin's Fee Market. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2950–2956. https://doi.org/10.1145/3308558.3313454

[77] Yoad Lewenberg, Yoram Bachrach, Yonatan Sompolinsky, Aviv Zohar, and Jeffrey S. Rosenschein. 2015. Bitcoin Mining Pools: A Cooperative Game Theoretic Analysis. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems* (Istanbul, Turkey) *(AAMAS '15)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 919–927. http://dl.acm.org/citation.cfm?id=2772879.2773270

[78] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive Block Chain Protocols. In *Financial Cryptography and Data Security*, Rainer Bohme and Tatsuaki Okamoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 528–547. https://doi.org/10.1007/978-3-662-47854-7_33 arXiv:https://doi.org/10.1007/978-3-662-47854-7_33

[79] Sheng-Nan Li, Zhao Yang, and Claudio J. Tessone. 2020. Mining blocks in a row: A statistical study of fairness in Bitcoin mining. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*. IEEE, 1–4. https://doi.org/10.1109/ICBC48266.2020.9169436

[80] Sheng-Nan Li, Zhao Yang, and Claudio J. Tessone. 2020. Proof-of-Work cryptocurrency mining: a statistical approach to fairness. In *2020 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*. IEEE. https://doi.org/10.1109/icccworkshops49972.2020.9209934

[81] Yizhong Liu, Yiming Hei, Tongge Xu, and Jianwei Liu. 2020. An Evaluation of Uncle Block Mechanism Effect on Ethereum Selfish and Stubborn Mining Combined With an Eclipse Attack. *IEEE Access* PP (01 2020), 1–1. https://doi.org/10.1109/ACCESS.2020.2967861

[82] Sishan Long, Soumya Basu, and Emin Gün Sirer. 2022. Measuring Miner Decentralization in Proof-of-Work Blockchains. https://doi.org/10.48550/ARXIV.2203.16058

[83] Ayelet Lotem, Sarah Azouvi, Patrick McCorry, and Aviv Zohar. 2022. Sliding Window Challenge Process for Congestion Detection. https://doi.org/10.48550/ARXIV.2201.09009

[84] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[85] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[86] Dmitry Meshkov, Alexander Chepurnoy, and Marc Jansen. 2017. Short paper: Revisiting difficulty control for blockchain systems. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 429–436. https://doi.org/10.1007/978-3-319-67816-0_25

[87] Johnnatan Messias, Mohamed Alzayat, Balakrishnan Chandrasekaran, Krishna P. Gummadi, Patrick Loiseau, and Alan Mislove. 2021. Selfish & Opaque Transaction Ordering in the Bitcoin Blockchain: The Case for Chain Neutrality. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 320–335. https://doi.org/10.1145/3487552.3487823

[88] Miningpoolstats. 2022. Ethereum Mining Pools. https://miningpoolstats.stream/ethereum

[89] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[90] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press.

[91] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. 2016. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 305–320. https://doi.org/10.1109/EuroSP.2016.32

[92] Rafael Pass and Elaine Shi. 2017. Fruitchains: A fair blockchain. In *Proceedings of the ACM symposium on principles of distributed computing*. 315–324.

[93] Julien Piet, Jaiden Fairoze, and Nicholas Weaver. 2022. Extracting Godl [sic] from the Salt Mines: Ethereum Miners Extracting Value. *arXiv preprint arXiv:2203.15930* (2022).

[94] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[95] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2021. Quantifying Blockchain Extractable Value: How dark is the forest? arXiv:2101.05511 [cs.CR]

[96] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2020. Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit. https://doi.org/10.1007/978-3-662-64322-8_1 arXiv:2003.03810 [cs.CR]

[97] BitMEX Research. 2022. ETHPoW vsETH2. https://web.archive.org/web/20220805155642/https://blog.bitmex.com/ethpow-vs-eth2/

[98] F. Ritz and A. Zugenmaier. 2018. The Impact of Uncle Rewards on Selfish Mining in Ethereum. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE Computer Society, Los Alamitos, CA, USA, 50–57. https://doi.org/10.1109/EuroSPW.2018.00013

[99] Matteo Romiti, Aljosha Judmayer, Alexei Zamyatin, and Bernhard Haslhofer. 2019. A deep dive into bitcoin mining pools: An empirical analysis of mining shares. *arXiv preprint arXiv:1905.05999* (2019).
[100] M. Rosenfeld. 2011. Analysis of Bitcoin Pooled Mining Reward Systems. *ArXiv e-prints* (Dec. 2011). arXiv:1112.4980 [cs.DC]
[101] Tim Roughgarden. 2020. Transaction Fee Mechanism Design for the Ethereum Blockchain: An Economic Analysis of EIP-1559. *CoRR* abs/2012.00854 (2020). arXiv:2012.00854 https://arxiv.org/abs/2012.00854
[102] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. 2016. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 515–532. https://doi.org/10.1007/978-3-662-54970-4_30 arXiv:1507.06183
[103] Afri Schoedon. 2018. EIP-1234: Constantinople Difficulty Bomb Delay and Block Reward Adjustment. https://eips.ethereum.org/EIPS/eip-1234
[104] Afri Schoedon and Vitalik Buterin. 2017. EIP-649: Metropolis Difficulty Bomb Delay and Block Reward Reduction. https://eips.ethereum.org/EIPS/eip-649
[105] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing Safe Smart Contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 218–219. https://doi.org/10.1145/3191697.3213790
[106] Ning Shi. 2016. A new proof-of-work mechanism for bitcoin. *Financial Innovation* 2, 1 (2016), 31. https://doi.org/10.1186/s40854-016-0045-6
[107] Paulo Silva, David Vavricka, João Barreto, and Miguel Matos. 2020. Impact of Geo-Distribution and Mining Pools on Blockchains: A Study of Ethereum. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 245–252. https://doi.org/10.1109/DSN48063.2020.00041
[108] Siamak Solat and Maria Potop-Butucaru. 2017. Brief Announcement: ZeroBlock: Timestamp-Free Prevention of Block-Withholding Attack in Bitcoin. In *Stabilization, Safety, and Security of Distributed Systems*, Paul Spirakis and Philippas Tsigas (Eds.). Springer International Publishing, Cham, 356–360.
[109] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *Financial Cryptography*.
[110] Yonatan Sompolinsky and Aviv Zohar. 2016. Bitcoin's Security Model Revisited. arXiv:1605.09193 [cs.CR]
[111] Yonatan Sompolinsky and Aviv Zohar. 2018. Bitcoin's underlying incentives. *Commun. ACM* 61 (02 2018), 46–53. https://doi.org/10.1145/3152481
[112] Douglas R Stinson. 2005. *Cryptography: theory and practice*. Chapman and Hall/CRC.
[113] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[114] Pawel Szalachowski. 2018. (Short Paper) Towards More Reliable Bitcoin Timestamps. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, IEEE, 101–104. https://doi.org/10.1109/cvcbt.2018.00018
[115] Lylian Teng. 2019. F2Pool Founder Condemns Block Withholding Attacks Performed by Some Chinese Mining Pools on Its Competitors. https://web.archive.org/web/20220214222028/https://news.8btc.com/f2pool-founder-condemns-block-withholding-attacks-performed-by-some-chinese-mining-pools-on-its-competitors
[116] The SageMath Developers. 2022. SageMath. https://doi.org/10.5281/zenodo.593563
[117] Itay Tsabary, Matan Yechieli, Alex Manuskin, and Ittay Eyal. 2020. MAD-HTLC: Because HTLC is Crazy-Cheap to Attack. https://doi.org/10.48550/ARXIV.2006.12031
[118] Guido Van Rossum et al. 2007. Python Programming language.. In *USENIX annual technical conference*, Vol. 41. 1–36.
[119] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2
[120] Luqin Wang and Yong Liu. 2015. Exploring Miner Evolution in Bitcoin Network. 290–302. https://doi.org/10.1007/978-3-319-15509-8_22
[121] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. 2019. A survey on consensus mechanisms and mining strategy management in blockchain networks. *IEEE* 7 (2019), 22328–22370. https://doi.org/10.1109/ACCESS.2019.2896108 arXiv:1805.02707
[122] Xu Wang, Xuan Zha, Guangsheng Yu, Wei Ni, Ren Ping Liu, Y Jay Guo, Xinxin Niu, and Kangfeng Zheng. 2018. Attack and defence of ethereum remote apis. In *2018 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–6.
[123] Ziyu Wang, Jianwei Liu, Qianhong Wu, Yanting Zhang, Hui Yu, and Ziyu Zhou. 2019. An Analytic Evaluation for the Impact of Uncle Blocks by Selfish and Stubborn Mining in an Imperfect Ethereum Network. *Comput. Secur.* 87, C (nov

2019), 10 pages. https://doi.org/10.1016/j.cose.2019.101581
[124] Sam M. Werner, Dragos I. Ilie, Iain Stewart, and William J. Knottenbelt. 2020. Unstable Throughput: When the Difficulty Algorithm Breaks. https://doi.org/10.1109/icbc51069.2021.9461086 arXiv:2006.03044 [cs.CR]
[125] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[126] Aviv Yaish. 2022. *Ethereum Scraper*. https://doi.org/10.6084/m9.figshare.20408058
[127] Aviv Yaish, Saar Tochner, and Aviv Zohar. 2022. Blockchain Stretching & Squeezing: Manipulating Time for Your Best Interest. In *Proceedings of the 23rd ACM Conference on Economics and Computation* (Boulder, CO, USA) *(EC '22)*. Association for Computing Machinery, New York, NY, USA, 65–88. https://doi.org/10.1145/3490486.3538250
[128] Aviv Yaish and Aviv Zohar. 2020. Pricing ASICs for Cryptocurrency Mining. arXiv:2002.11064
[129] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 349–365. https://www.usenix.org/conference/osdi21/presentation/yang
[130] Liyi Zeng, Yang Chen, Shuo Chen, Xian Zhang, Zhongxin Guo, Wei Xu, and Thomas Moscibroda. 2021. Characterizing Ethereum's Mining Power Decentralization at a Deeper Level. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications* (Vancouver, BC, Canada). IEEE Press, 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488812
[131] Ren Zhang and Bart Preneel. 2017. Publish or perish: A backward-compatible defense against selfish mining in bitcoin. In *Cryptographers' Track at the RSA Conference*. Springer, 277–292.
[132] Ren Zhang and Bart Preneel. 2019. Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, 175–192. https://doi.org/10.1109/sp.2019.00086
[133] Roi Bar Zur, Ameer Abu-Hanna, Ittay Eyal, and Aviv Tamar. 2022. WeRLman: To Tackle Whale (Transactions), Go Deep (RL). *IACR Cryptol. ePrint Arch.* (2022), 175. https://eprint.iacr.org/2022/175
[134] Roi Bar Zur, Ittay Eyal, and Aviv Tamar. 2020. Efficient MDP analysis for selfish-mining in blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. ACM, 113–131. https://doi.org/10.1145/3419614.3423264

A APPENDICES STRUCTURE

Due to a lack of space, we have omitted examples, proofs, extensions of our attack and various other details from the main body of the paper, but included them as appendices.

The appendices are structure as follows: in Appendix B we provide an algorithmic description of the RUM attack, and then in Appendix C we go over a real-world series of blocks which we suspect to be a real-life attempt at executing the attack. Appendix D contains the proofs for our theoretical analysis of the RUM attack. Appendix E attempts to paint a bigger picture of the current research landscape by going over papers which were not included in Section 9. In Appendix F we provide all necessary information in order to reproduce our work, such as the code we used to extract data from the blockchain and create the various graphs included in the paper, a link to an anonymous website which contains a copy of our data and alternative sources of data. Finally, Appendix G is a glossary of the various notations and acronyms used throughout the paper.

B ALGORITHM

We provide an algorithmic description of the RUM attack in Algorithm 1.

C REAL-WORLD EXAMPLE

Example 2. *We will go over a real-world counterpart for Example 1. We follow Ethereum's conventions and use the hexadecimal numeral system when referring to various values such as block hashes and difficulties.*

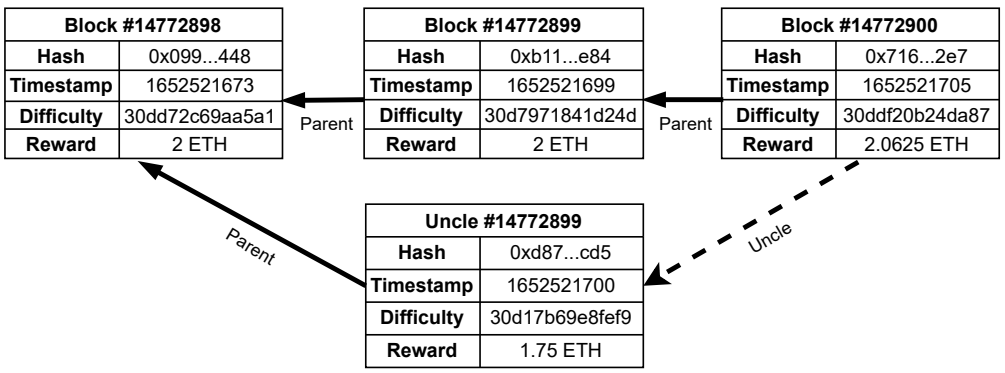

Fig. 14. A depiction of the blocks of Example 2.

Fig. 14 depicts four real blocks. The block with hash $0x716...2e7$ *(abbreviated for brevity) has a depth of* 14772900 *blocks relative to the genesis. This block references the block with hash* $0xb11...e84$ *as its parent, and the block with hash* $0xd87...cd5$ *as its uncle. Both the parent and the uncle reference block number* 14772898 *as their parent, which we will refer to as the grandparent, for clarity.*

The timestamp difference between the parent and the grandparent is 26 *seconds, while the difference between the uncle and the grandparent is* 27 *seconds. Thus, according to Ethereum's DAA, the parent was mined at a higher difficulty.*

The grandparent has a TD of $a5cd3e87d79bc358c2d$*. As the TD is a simple sum of the each block's difficulty and its direct ancestor's TD, the parent's TD is* $a5cd4195510d4775e7a$*, while the uncle's is lower at* $a5cd4194ef5261e8b26$*. So, according to Ethereum's tie breaking rule, the former* must *be the current block's parent.*

Notice that block $0xb11...e84$ *was mined by F2Pool.*

Algorithm 1: Riskless uncle maker attack

```
on event initialize:
    chain ← publicly known blocks ;
    do: mine honestly on top of the tip of chain ;
end
on event we mined a block b:
    do: publish b and append it to chain ;
    do: mine honestly on top of the tip of chain ;
end
on event others mined a block b:
    if t_b ∈ [9, 18) then
        trigger event: attack against b ;
    else
        do: append b to chain ;
        do: mine honestly on top of the tip of chain ;
    end
end
on event attack against b:
    t_H ← 0 ;
    while t_H < 9 & no-one mined a new block do
        mine b_A with t_{b_A} = 8 on top of b.parent ;
        t_H ← currentTime − b.timestamp ;
    end
    if honest miners mined a new block b′ then
        do: append b to chain ;
        trigger event: others mined a block b′ ;
    else if we mined b_A then
        trigger event: we mined a block b_A ;
    else
        do: append b to chain ;
        do: mine honestly on top of the tip of chain ;
    end
end
```

D PROOFS

D.1 Conditions For a Riskless Uncle Maker Attack

We will now characterize the conditions in which a RUM attack is feasible.

By executing the attack, the attacker increases its share of the total rewards and of the total main-chain blocks in the network, above its fair share. In addition, the attack yields the same number of block rewards as honest mining would.

Theorem 1. *Let the current tip of the main-chain be block b_1. Denote its parent as b_0, the parent's difficulty as d_0, the timestamp difference between them as t_1, and the difference between the current time and b_1's timestamp as t_H.*

If the following conditions hold, then a rational attacker can execute a RUM attack:

$$\lfloor \frac{t_H}{9} \rfloor = 0,\ \lfloor \frac{t_1}{9} \rfloor = 1$$

Moreso, if $d_0 \geq 2^{22}$, these are the only values for which a RUM attack is possible.

An attacker needs to know all the parameters required by Theorem 1 in order to evaluate whether a riskless attack is currently possible. Remark 1 explains how these values can be obtained.

In order to prove Theorem 1, we break the attack down to its different constituents: the blocks generated by the attacker should be valid according to the system's consensus rules, the attacker should successfully replace honest blocks with the attacker's blocks, and mining these blocks should be *riskless*, meaning that it should not be harder when compared to mining honestly. Each constituent will be handled via a series of claims.

Given the setting of Theorem 1, the honest miner is currently mining a block on top of b_1, per the honest protocol. Denote this pending block as b_H, and its relative timestamp difference as t_H.

In order to "kick-out" b_1 from the main-chain, the attacker needs to mine a block on top of some main-chain block which precedes b_1. The latest block which fulfills this criterion is b_0. According to our threat model, the attacker can mine on top of b_0, and can also set the timestamp arbitrarily, as long as it's valid. Denote the attacker's pending block by b_A, define its parent as b_0, and set its timestamp difference to be t_A, which we will soon define.

A visual depiction of the current state of the blockchain and all pending blocks is provided in Fig. 2.

All the conditions that the attacker's block must fulfil are:

(1) The block should be valid. Concretely, the timestamp difference between b_A and b_0 should be valid:

$$t_A \geq 1 \tag{2}$$

(2) The block should win ties with b_1.
(3) Mining the block should not be harder when compared to mining honestly: $d_A \leq d_H$.

Note that the third condition is only required for riskless attacks. We will now use a series of claims to find timestamps that answer these requirements.

Claim 1. *Block b_A will win ties against b_1 if:*

$$\lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right)$$

Proof. According to Ethereum's tie-breaking mechanism, in case of ties, blocks with a higher TD are preferred. As both blocks have the same parent, and because TD is a simple sum of a block's difficulty and its parent's difficulty, this requirement boils down to:

$$d_A > d_1 \tag{3}$$

Thus, by plugging in Eq. (1):

$$d_0 + \max\left(1 + 0 - \lfloor \frac{t_A}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor > d_0 + \max\left(1 + 0 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \tag{4}$$

Slight algebraic manipulations give us:

$$\max\left(1 - \lfloor \frac{t_A}{9} \rfloor, -99\right) > \max\left(1 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \tag{5}$$

Note that by assumption $\lfloor \frac{t_A}{9} \rfloor < 100$, and thus it is always the case that $1 - \lfloor \frac{t_A}{9} \rfloor > -99$. This means that the above condition is equivalent to:

$$1 - \lfloor \frac{t_A}{9} \rfloor > \max\left(1 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \tag{6}$$

First, we handle the case where the right term obtains a maximum at -99. This is equivalent to:

$$1 - \lfloor \frac{t_A}{9} \rfloor > -99 \tag{7}$$

As noted above, this is always true, meaning that the adversary will win ties in this case.

In the other case, if the right term of Eq. (6) obtains a maximum at $1 - \lfloor \frac{t_1}{9} \rfloor$, we can rewrite the equation like so:

$$1 - \lfloor \frac{t_A}{9} \rfloor > 1 - \lfloor \frac{t_1}{9} \rfloor \tag{8}$$

By simplifying this equation, we get:

$$\lfloor \frac{t_A}{9} \rfloor < \lfloor \frac{t_1}{9} \rfloor \tag{9}$$

This inequality also holds by assumption, meaning that the adversary wins ties in this case as well. □

Now, we will analyze the opposite scenario.

Claim 2. *Block b_A will lose ties against b_1 if:*

$$\lfloor \frac{t_A}{9} \rfloor \geq \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right)$$

Proof. The difficulties of the blocks are:

$$d_A \stackrel{\text{def}}{=} d_0 + \max\left(1 + 0 - \lfloor \frac{t_A}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \tag{10}$$

$$d_1 \stackrel{\text{def}}{=} d_0 + \max\left(1 + 0 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \tag{11}$$

By our assumption that $\lfloor \frac{t_A}{9} \rfloor \geq \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right)$, we get:

$$\begin{aligned} d_A = d_0 + \max\left(1 + 0 - \lfloor \frac{t_A}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \leq \\ d_0 + \max\left(1 + 0 - \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right), -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = \\ d_0 + \max\left(1 - \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right), -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = \\ d_0 + \max\left(1 - \lfloor \frac{t_1}{9} \rfloor, 1 - 100, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = \\ d_0 + \max\left(1 - \lfloor \frac{t_1}{9} \rfloor, -99, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = \\ d_0 + \max\left(1 + 0 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = d_1 \end{aligned} \tag{12}$$

As the attacker will start mining b_A only *after* the honest miner has published b_1, by definition it is impossible for the honest miner to hear about b_A before b_1.

Following Ethereum's consensus rules as defined in Section 4.1, this means that the honest miner will prefer b_A only if its difficulty is higher than b_1. As Eq. (12) shows, this is not the case. □

Claims 1 and 2 allow us to rule out the possibility of performing a successful attack in certain cases.

Corollary 2. *A RUM attack cannot be performed if $t_1 < 9$.*

Proof. We know from Claims 1 and 2 that b_1 can be replaced iff:

$$\lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) \tag{13}$$

But, the attacker's timestamp must be valid, e.g. must satisfy Eq. (2):

$$1 \leq \lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) \tag{14}$$

If $t_1 < 9$, then:

$$1 \leq \lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) = 0 \tag{15}$$

And we've reached a contradiction. □

Now, in order to guarantee that the attack is indeed "riskless", we must make sure that the probability of the attacker successfully mining on top of b_0 is not lower than if it were following the honest protocol and mining on top of b_1. This is taken care of by Claim 3.

Claim 3. *Assuming $\lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right)$ and $t_1 \geq 9$, block b_A incurs no additional mining risk compared to mining honestly on top of b_1 if:*

$$\lfloor \frac{t_A}{9} \rfloor \geq \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) + \min\left(\lfloor \frac{t_H}{9} \rfloor - 1, 99\right) \frac{\lfloor \frac{d_1}{2048} \rfloor}{\lfloor \frac{d_0}{2048} \rfloor}$$

Proof. In order to guarantee that the attack is riskless, we must make sure that the difficulty that corresponds to the attacker's time difference is not higher than the one used by the honest miner's pending block:

$$d_A \leq d_H \tag{16}$$

By plugging in Eq. (1):

$$d_0 + \max\left(1 + 0 - \lfloor \frac{t_A}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \leq d_1 + \max\left(1 + 0 - \lfloor \frac{t_H}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_1}{2048} \rfloor \tag{17}$$

From our assumptions, we know that:

$$-\min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) < -\lfloor \frac{t_A}{9} \rfloor \tag{18}$$

So:

$$1 - \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) < 1 - \lfloor \frac{t_A}{9} \rfloor \tag{19}$$

According to the consensus rules defined in Section 4, we know that $t_1 \geq 1$. By plugging the maximal value of 100 into the previous equation, we can deduce:

$$1 - 100 = -99 < 1 - \lfloor \frac{t_A}{9} \rfloor \tag{20}$$

This allows us to simplify Eq. (17) to:

$$d_0 + \left(1 - \lfloor \frac{t_A}{9} \rfloor\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \leq d_1 + \max\left(1 - \lfloor \frac{t_H}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_1}{2048} \rfloor \tag{21}$$

By plugging in d_1 according to the definition given in Eq. (1):

$$d_0 + \left(1 - \lfloor \frac{t_A}{9} \rfloor\right) \cdot \lfloor \frac{d_0}{2048} \rfloor \leq d_0 + \max\left(1 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor + \max\left(1 - \lfloor \frac{t_H}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_1}{2048} \rfloor \tag{22}$$

Slightly rearranging, we get:

$$\left(1 - \lfloor\frac{t_A}{9}\rfloor - \max\left(1 - \lfloor\frac{t_1}{9}\rfloor, -99\right)\right) \cdot \lfloor\frac{d_0}{2048}\rfloor \leq \max\left(1 - \lfloor\frac{t_H}{9}\rfloor, -99\right) \cdot \lfloor\frac{d_1}{2048}\rfloor \tag{23}$$

By applying simple algebra:

$$\max\left(1 - \lfloor\frac{t_H}{9}\rfloor, -99\right) \cdot \frac{\lfloor\frac{d_1}{2048}\rfloor}{\lfloor\frac{d_0}{2048}\rfloor} \geq 1 - \lfloor\frac{t_A}{9}\rfloor - \max\left(1 - \lfloor\frac{t_1}{9}\rfloor, -99\right) \tag{24}$$

We would like to isolate $\lfloor\frac{t_A}{9}\rfloor$:

$$\lfloor\frac{t_A}{9}\rfloor \geq 1 - \max\left(1 - \lfloor\frac{t_1}{9}\rfloor, -99\right) - \max\left(1 - \lfloor\frac{t_H}{9}\rfloor, -99\right) \cdot \frac{\lfloor\frac{d_1}{2048}\rfloor}{\lfloor\frac{d_0}{2048}\rfloor} \tag{25}$$

Finally:

$$\lfloor\frac{t_A}{9}\rfloor \geq \min\left(\lfloor\frac{t_1}{9}\rfloor, 100\right) + \min\left(\lfloor\frac{t_H}{9}\rfloor - 1, 99\right) \frac{\lfloor\frac{d_1}{2048}\rfloor}{\lfloor\frac{d_0}{2048}\rfloor} \tag{26}$$

□

In Example 3 we will show a specific setting in which the mining difficulties of b_A and b_H are equal.

Example 3. *Let $t_1 \in [9, 18)$ and $t_H < 9$. According to Claims 1 to 3, an attacker can attempt to perform a RUM attack by mining b_A with a timestamp difference which satisfies $t_A \in [1, 9)$.*

By plugging these values in Eq. (1)*, we get that the mining difficulties of b_1 and b_0 are equal:*

$$\begin{aligned} d_1 &= d_0 + \left(1 - \lfloor\frac{t_1}{9}\rfloor\right) \cdot \lfloor\frac{d_0}{2048}\rfloor \\ &= d_0 + (1-1) \cdot \lfloor\frac{d_0}{2048}\rfloor \\ &= d_0 + 0 \cdot \lfloor\frac{d_0}{2048}\rfloor \\ &= d_0 \end{aligned} \tag{27}$$

Thus, the mining difficulties of b_A and b_H are equal too:

$$\begin{aligned} d_A &= d_0 + \left(1 - \lfloor\frac{t_A}{9}\rfloor\right) \cdot \lfloor\frac{d_0}{2048}\rfloor \\ &= d_0 + (1-0) \cdot \lfloor\frac{d_0}{2048}\rfloor \\ &= d_1 + (1-0) \cdot \lfloor\frac{d_1}{2048}\rfloor \\ &= d_1 + \max\left(1 - \lfloor\frac{t_H}{9}\rfloor, -99\right) \cdot \lfloor\frac{d_1}{2048}\rfloor \\ &= d_H \end{aligned} \tag{28}$$

This example proves the first part of Theorem 1, showing that a rational attacker can execute a RUM attack if $\lfloor\frac{t_A}{9}\rfloor = 0, \lfloor\frac{t_1}{9}\rfloor = 1$. The proof of the theorem is concluded using Claims 1 to 3.

Theorem 1. *Let the current tip of the main-chain be block b_1. Denote its parent as b_0, the parent's difficulty as d_0, the timestamp difference between them as t_1, and the difference between the current time and b_1's timestamp as t_H.*

If the following conditions hold, then a rational attacker can execute a RUM attack:

$$\lfloor \frac{t_H}{9} \rfloor = 0,\ \lfloor \frac{t_1}{9} \rfloor = 1$$

Moreso, if $d_0 \geq 2^{22}$, these are the only values for which a RUM attack is possible.

Proof. By combining Claims 1 to 3, we get:

$$\min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) + \min\left(\lfloor \frac{t_H}{9} \rfloor - 1, 99\right) \frac{\lfloor \frac{d_1}{2048} \rfloor}{\lfloor \frac{d_0}{2048} \rfloor} \leq \lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) \tag{29}$$

Note that when $\lfloor \frac{t_H}{9} \rfloor > 0$, we get $\min\left(\lfloor \frac{t_H}{9} \rfloor - 1, 99\right) \geq 0$, meaning that:

$$\min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) + \min\left(\lfloor \frac{t_H}{9} \rfloor - 1, 99\right) \frac{\lfloor \frac{d_1}{2048} \rfloor}{\lfloor \frac{d_0}{2048} \rfloor} \geq \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) \tag{30}$$

This means that no t_A exists which can satisfy Eq. (29). Therefore, a suitable t_A can only exist when $\lfloor \frac{t_H}{9} \rfloor = 0$, and thus $t_H < 9$. Assuming that this holds, we get:

$$\min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) - \frac{\lfloor \frac{d_1}{2048} \rfloor}{\lfloor \frac{d_0}{2048} \rfloor} \leq \lfloor \frac{t_A}{9} \rfloor < \min\left(\lfloor \frac{t_1}{9} \rfloor, 100\right) \tag{31}$$

In order for there to be an integer between the two bounds, the following must hold as well:

$$\lfloor \frac{d_1}{2048} \rfloor \geq \lfloor \frac{d_0}{2048} \rfloor \tag{32}$$

According to Eq. (1), this can only hold if:

$$\lfloor \frac{d_0 + \max\left(1 + 0 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor}{2048} \rfloor \geq \lfloor \frac{d_0}{2048} \rfloor \tag{33}$$

This equation holds when $\lfloor \frac{t_1}{9} \rfloor \in \{0, 1\}$.

We would now like to show that these conditions are necessary when $d_0 \geq 2^{22} = 2048^2$. If $\lfloor \frac{t_1}{9} \rfloor \geq 2$ and $d_0 \geq 2048^2$, then we get:

$$\begin{aligned} \lfloor \frac{d_1}{2048} \rfloor &= \lfloor \frac{d_0 + \max\left(1 + 0 - \lfloor \frac{t_1}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor}{2048} \rfloor \\ &\leq \lfloor \frac{d_0 - \lfloor \frac{2048^2}{2048} \rfloor}{2048} \rfloor \\ &= \lfloor \frac{d_0 - 2048}{2048} \rfloor \\ &= \lfloor \frac{d_0}{2048} \rfloor - 1 \end{aligned} \tag{34}$$

Finally, a contradiction is reached when substituting the above into Eq. (32):

$$\lfloor \frac{d_0}{2048} \rfloor \leq \lfloor \frac{d_1}{2048} \rfloor \leq \lfloor \frac{d_0}{2048} \rfloor - 1 \tag{35}$$

As shown in Corollary 2, the attack is not possible when $\lfloor \frac{t_1}{9} \rfloor = 0$, meaning that attack is only possible when $\lfloor \frac{t_1}{9} \rfloor = 1$, as required. □

D.2 Absolute Profits of a Riskless Attack

Our analysis follows the notations laid in Section 5. Let b_1 be the tip of of the main chain with difficulty d_1 and b_0 be its parent with difficulty d_0. In addition, let t_1 be the difference between b_1 and b_0's timestamps. Finally, define $p_{t<9}$ to be the probability that a block is mined on top of b_0 with time difference $t < 9$ in 9 seconds with the entire compute-power of the network. Similarly, let $p_{9\leq t<18}$ to be the probability that a block is mined on top of b_0 with time difference $9 \leq t < 18$ in 9 seconds with the entire compute-power of the network.

If $9 \leq t_1 < 18$, then according to Ethereum's DAA, given in Eq. (1): $d_1 = d_0 - (1-1)\left\lfloor \frac{d_0}{2048} \right\rfloor = d_0$. Therefore, the probabilities $p_{t<9}, p_{9\leq t<18}$ associated with mining on top of either b_0 or b_1 are identical, as shown in the proof of Theorem 1.

In the described case, the attacker tries to mine on top of b_0, setting its timestamp to be between 1 and 8 seconds later than b_0's. In other words, if we define b_A to be the block currently mined by the attacker, and t_A to be the difference between b_0's timestamp and b_A's timestamp, then $0 < t_A < 9$. Computing the attacker's difficulty in the first 9 seconds of the attack we get:

$$d_A = d_0 + \max\left(1 + 0 - \lfloor \frac{t_A}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_0}{2048} \rfloor = d_0 + \lfloor \frac{d_0}{2048} \rfloor \tag{36}$$

On the other hand, mining honestly on top of b_1 for the first 9 seconds after it was mined, i.e. with $0 < t_H < 9$, yields a difficulty of:

$$d_H = d_1 + \max\left(1 + 0 - \lfloor \frac{t_H}{9} \rfloor, -99\right) \cdot \lfloor \frac{d_1}{2048} \rfloor = d_0 + \lfloor \frac{d_0}{2048} \rfloor \tag{37}$$

This means that the attacker mining on top of b_0 and the honest network mining on top of b_1, mine with the same difficulty for the first 9 seconds. As shown above, this difficulty is exactly the difficulty of mining on top of b_0 in the first 9 seconds. In terms of the probability $p_{t<9}$, the adversary's probability of successfully mining a block during the attack is $p_{t<9}\alpha$.

Since the adversary's behaviour is identical in all cases but the 9 seconds in which it perform the attack, all of its rewards would be identical all of those cases. However, if the adversary manages to mine honestly on top of b_1, it would receive a single block reward R for its efforts. On the other hand, if its attack succeeds and it mines on top of b_0 instead, it would receive a single block reward, and all honest nodes would mine on top of the adversary's block, possibly including b_1 as an uncle.

This leads us to formulating a theorem, showing that the adversary can receive a larger share of the rewards, while not reducing the absolute amount rewards.

Theorem 3. *For any block b_0, an attacker can increase its expected absolute and relative rewards by using the RUM mining strategy instead of mining honestly.*

Proof. Let A be the amount of ETH the adversary has before performing the attack and H be the amount of ETH the honest network has at that time. If the adversary acts honestly and manages to mine on top of b_1 it receives R ETH and F ETH in fees, and the honest network receives R ETH and F ETH as well for mining b_1. On the other hand, if the adversary performs the attack and manages to mine on top of b_0 in fewer than 9 seconds, it receives R ETH as well as F ETH in fees, and the honest network receives at most $\frac{7}{8}R$ ETH as a result of b_1 being added as an uncle. Note that in both cases, the attacker has $A + R + F$ ETH after successfully performing the attack. This means that with probability $p_{t<9}\alpha$, A can increase its share of the total ETH from $\frac{A+R+F}{A+H+2R+2F}$ to at least $\frac{A+R+F}{A+H+(1+\frac{7}{8})R+F}$. Since in all other cases, the attacker mines honestly, it receives the same rewards and has the same share of the total ETH in the network. □

We can also formulate a similar theorem, describing the adversary's share and total number of main-chain blocks.

Theorem 4. *For any block b_0, the attacker mines the same number of main-chain blocks when performing the attack as it would mining honestly. Furthermore, the attacker mines a larger share of the number of main-chain blocks than it would mining honestly.*

Proof. Using similar analysis, now define A to be the number of total main-chain blocks controlled by the adversary before performing the attack and H to be the number of total main-chain blocks controlled by the honest network. If the attacker mines a block honestly on top of b_1, then its share of the total number of main-chain blocks would be $\frac{A+1}{A+H+2}$. On the other hand, if the attacker performs the attack and mines successfully on top of b_0 instead, it would increase its share of the total number of main-chain blocks to $\frac{A+1}{A+H+1}$, because b_1 would be relegated to an uncle instead of a main-chain block. This event takes place with probability $p_{t<9}\alpha$. In all other events, the attacker acts honestly and receives the same number and share of total main-chain blocks in the system. □

A more formal analysis of the expected share of the rewards is provided in Appendix D.5 and an analysis of the expected share of main-chain blocks is provided in Appendix D.4.

D.3 Markov Decision Process for the Riskless Uncle Maker Attack

Our next proofs rely on analyzing MDPs, thus we will first define what MDPs are.

D.3.1 Markov Decision Processes. Multi-agent stochastic processes in discrete time are commonly modeled in the blockchain literature using Markov decision processes [50, 59, 67, 102, 127, 133, 134]. We, too, will use this technique to analyze our attack.

A MDP can be defined using a tuple of four elements:

$$(S, A_s, P_a(\cdot,\cdot), R_a(\cdot,\cdot))$$

With these elements in turn defined as follows [94]:

(1) S is the set of all possible world *states*.
(2) A_s is the set of *actions* which can be taken in state s.
(3) $P_a(s, s')$ is the probability of *transitioning* from state $s \in S$ to $s' \in S$ given that the action $a \in A_s$ was taken at state $s \in S$.
(4) $R_a(s, s')$ is the *reward* earned by taking action $a \in A_s$ in state $s \in S$ and transitioning to s'.

In order to model a discrete-time random process which is *memoryless*, e.g. the next state s_{t+1} relies only on the current state s_t and current action a_t, one can define P_a like so:

$$P_a(s, s') \stackrel{\text{def}}{=} P(s_{t+1} = s' | s_t = s, a_t = a)$$

This memoryless-ness is also known as the *Markov property*.

Finding a mapping between states and reward-maximizing actions for some specific MDP is commonly called *solving* the MDP. Although we will use MDPs throughout our work, solving them constitutes an academic field in itself and is not the main focus of our work. We would like to turn diligent readers wishing to deepen their knowledge on MDPs to classic texts such as [51, 94, 113].

D.3.2 A Markov Decision Process for the Attack. We now proceed to describe the MDP we will use to analyze the expected profits that a RUM attack can produce. A state-diagram of this MDP is provided in Fig. 15, and a tabular description is given in Table 2.

D.3.3 State Space and Actions. The MDP has two possible states:

(1) *Normal*. In this state, the conditions for a RUM attack, as defined in Theorem 1, are not fulfilled. Thus, the attacker mines normally, e.g. using the honest protocol.
(2) *Attack*. This state corresponds to a world-state at which the conditions for the RUM attack are present. In this state, the attacker mines according to the strategy specified in Section 5.

Table 2. The MDP for the theoretical analysis of our attack, in tabular form.

State	Transition	Destination	Probability	Blocks	
				Attacker	Honest
Normal	Attacker at any time	Normal	α	1	0
	Honest $t < 9$	Normal	$p_{t<9} \cdot (1-\alpha)$	0	1
	Honest $t \geq 18$	Normal	$(1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)$	0	1
	Honest $9 \leq t < 18$	Attack	$(1-p_{t<9}) \cdot p_{9\leq t<18} \cdot (1-\alpha)$	0	0
Attack	Attacker $t < 9$	Normal	$p_{t<9} \cdot \alpha$	1	0
	Attacker $t \geq 9$	Normal	$(1-p_{t<9})\alpha$	1	1
	Honest $t < 9$ or $18 \leq t$	Normal	$p_{t<9} \cdot (1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)$	0	2
	Honest $9 \leq t < 18$	Attack	$(1-p_{t<9}) \cdot p_{9\leq t<18} \cdot (1-\alpha)$	0	1

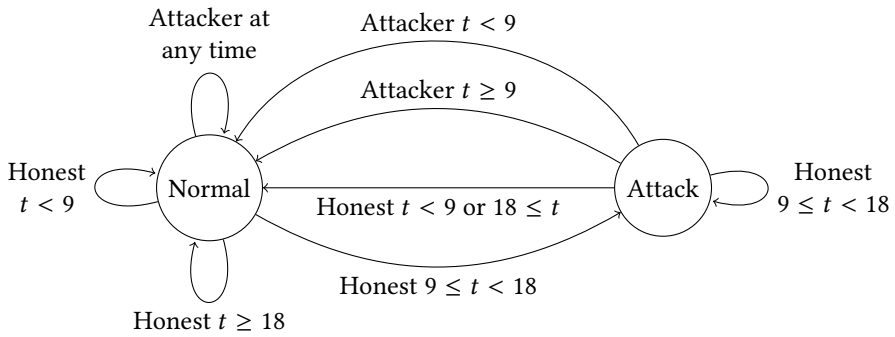

Fig. 15. The MDP for the theoretical analysis of our attack, as a state diagram.

D.3.4 State Transitions. Transitions from the *Normal* state to *Attack* occur when the conditions specified in Theorem 1 are present. Due to Remark 2, these conditions boil down to a single possibility: an attacker can execute the RUM attack if the tip of the blockchain was mined by the honest party, and has a timestamp difference t from the preceding block such that $9 \leq t < 18$. Thus, we will call this transition *Honest* $9 \leq t < 18$.

If we denote by $p_{t<9}$ the probability that a block was mined in less than 9 seconds after its parent, and the probability that a block was mined between 9 (inclusive) and 18 seconds after its parent as $p_{9\leq t<18}$, then the probability of transitioning from the *Normal* state to the *Attack* state is equal to:

$$(1-p_{t<9}) \cdot p_{9\leq t<18} \cdot (1-\alpha) \tag{38}$$

Note that in all other cases, transitions exiting the *Normal* state will lead back to the *Normal* state:

(1) If the honest miner succeeded in mining a new block before the attacker, and the block's timestamp difference is not between 9 and 18. We will call the two possible transitions *Honest* $t < 9$ and *Honest* $t \geq 18$.
(2) If the attacker mined a new block before the honest miner. We will call this transition *Attacker at any time*.

We will now examine transitions that exit the *Attack* state by splitting them to the two possible cases:

A. The attack succeeded.

This can only happen if the attacker mines the attack block before the honest miner and in less than 9 seconds since starting the attack. We will call this transition *Attacker* $t < 9$.

B. The attack failed.
This case has multiple possibilities:

(1) The honest miner succeeded in mining another block before the attacker successfully mined the attack block.
If this new block has a timestamp difference which is between 9 and 18, the attacker can execute a new attack. So, although the current attack has failed, we remain in the *Attack* state. We will call this case *Honest* $9 \leq t < 18$, similarly to the corresponding transition which exits the *Normal* state.
We are left with two other possibilities: either the new block's timestamp difference is lower than 9, or higher than 18, leading us to call this transition *Honest* $t < 9$ *or* $18 \leq t$.
(2) The attacker succeeded in mining a block before the honest miner, but after the attack's 9 second time window has ended.

D.3.5 Transition Rewards. In order to analyze the attack's effect on both the amount of main-chain blocks and rewards, we used almost-identical MDPs which differ solely by the transition rewards.

In one, the rewards for transitioning between states are the number of blocks which each actor has mined, while in the other the rewards are the block-rewards earned by miners. For brevity, we will only go over the former.

The rewards are relatively straightforward for all transitions that start from *Normal* and move to *Normal*, as both the attacker and honest miner mine honestly. For example, when the transition *Attacker at any time* occurs, the attacker mines a single block and thus receives a reward of 1, while the honest miner receives 0.

Note that in the sole transition from *Normal* to *Attack*, both miners receive a reward of 0 although the honest miner *did* mine a block. This is done because if the attack succeeds, this block will be replaced by an attacker block. Thus, we account for the honest miner's block only in transitions which correspond to the attack failing:

(1) *Attacker* $t \geq 9$. In this transition, the attacker succeeded in mining a block after the attack's 9 second time window has ended, but did so before the honest miner had succeeded in mining an additional block.
Recall that our attack instructs the attacker to mine honestly after the attack's time window has ended, and thus the attacker's block was mined on top of the block it previously attempted to replace. So, we reward both the attacker and the honest miner with a single block.
(2) *Honest* $t < 9$ *or* $18 \leq t$. In both of these cases, the honest miner has succeeded in mining before the attacker, meaning that the attack has failed. Thus, the honest miner already accumulated two blocks which we have to account for, while the attacker did not mine even a single one.
(3) *Honest* $9 \leq t < 18$. In this case, the honest miner mined an additional block before the attacker, and this block has a timestamp difference (relative to its ancestor) which allows the attacker to execute a new RUM attack. Thus, we remain in the *Attack* state, but reward the honest miner for its previous block.

D.4 Expected Block Share

We will now show that the attacker's expected share of the blocks is larger than its fair share.

Theorem 2. *Let there be some block* b_0. *If the attacker uses the RUM mining strategy, its expected relative share of main-chain blocks will be larger than mining honestly, while the absolute number will remain the same.*

To prove this, we will solve the MDP.

Claim 4. *The stationary distribution is:*

$$P(Attack) = (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)$$
$$P(Normal) = 1 - P(Attack)$$

Proof. According to the MDP:

$$P(Attack) = (P(Attack) \cdot (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)) + (P(Normal) \cdot (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)) \tag{39}$$

Using simple algebra:

$$P(Attack) = (1 - p_{t<9})p_{9\leq t<18}(1-\alpha) \cdot (P(Attack) + P(Normal)) \tag{40}$$

As our only two state are *Attack* and *Normal*, we get:

$$P(Attack) + P(Normal) = 1 \tag{41}$$

By combining Eqs. (40) and (41):

$$P(Attack) = (1 - p_{t<9})p_{9\leq t<18}(1-\alpha) \cdot 1 \tag{42}$$

Finally, using Eqs. (41) and (42):

$$P(Normal) = 1 - P(Attack) \tag{43}$$

And we're done. □

We will use the stationary distribution to calculate the attacker's expected share of blocks.

Claim 5. *The attacker's expected number of blocks is:*

$$Attacker = \alpha$$

Proof. The attacker mines a single block in each of the following transitions:

(1) Started at the *Normal* state, and the event *Attacker at any time* occurred.
(2) Started at the *Attack* state, and any of the events *Attacker* $t < 9$, *Attacker* $t \geq 9$ occurred.

Thus its expected number of main-chain blocks is:

$$Attacker = P(Normal) \cdot \alpha + P(Attack) \cdot (p_{t<9}\alpha + (1 - p_{t<9})\,\alpha) \tag{44}$$

Using algebra and substituting for the state probabilities we get:

$$Attacker = \alpha \tag{45}$$

□

We will now do the same for the honest miner.

Claim 6. *The honest miner's expected number of blocks is:*

$$Honest = (1-\alpha) - (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)$$

Proof. It mines two blocks when starting at the *Attack* state and the event *Honest* $t < 9$ *or* $18 \leq t$ occurred, and mines a single block at each of the following transitions:

(1) When starting at the *Normal* state and either *Honest* $t < 9$ or *Honest* $t \geq 18$ occurred.
(2) When starting at the *Attack* state and *Honest* $9 \leq t < 18$ occurred.

After tallying up all events, we arrive at the following expected number of blocks:

$$\begin{aligned} Honest = {} & P\left(Normal\right) \cdot p_{t<9}\left(1-\alpha\right) + P\left(Normal\right) \cdot \left(1-p_{t<9}\right)\left(1-p_{9\leq t<18}\right)\left(1-\alpha\right) + \\ & P\left(Attack\right) \cdot 2\left(1-\alpha\right)\left(p_{t<9} + \left(1-p_{t<9}\right)\left(1-p_{9\leq t<18}\right)\right) + \\ & P\left(Attack\right)\left(\left(1-p_{t<9}\right)\alpha + \left(1-p_{t<9}\right)p_{9\leq t<18}\left(1-\alpha\right)\right) \end{aligned} \tag{46}$$

We begin by plugging-in both state probabilities:

$$\begin{aligned} Honest = {} & ((1-(1-p_{t<9})p_{9\leq t<18}(1-\alpha)) \cdot (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha))) \\ & + (2(1-p_{t<9})p_{9\leq t<18}(1-\alpha) \cdot (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha))) \\ & + ((1-p_{t<9})p_{9\leq t<18} \cdot (1-\alpha)((1-p_{t<9})\alpha + (1-p_{t<9})p_{9\leq t<18}(1-\alpha))) \end{aligned} \tag{47}$$

We proceed with simplifying the previous equation; the following is not for the faint of heart:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + ((1-p_{t<9})p_{9\leq t<18}(1-\alpha) \cdot \\ & ((1-p_{t<9})\alpha + (1-p_{t<9})p_{9\leq t<18}(1-\alpha))) + ((1-p_{t<9})p_{9\leq t<18}(1-\alpha) \cdot \\ & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha))) \end{aligned} \tag{48}$$

Using some algebra:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + \\ & (1-p_{t<9})p_{9\leq t<18}(1-\alpha)((1-p_{t<9})\alpha + (1-p_{t<9})(1-\alpha)) + (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}(1-\alpha)) \end{aligned} \tag{49}$$

By grouping together equal terms, we get:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + \\ & (1-p_{t<9})p_{9\leq t<18}(1-\alpha)((1-p_{t<9}) + p_{t<9}(1-\alpha)) \end{aligned} \tag{50}$$

We will now add and remove the term $(1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)$, this will help later on:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + (1-p_{t<9})p_{9\leq t<18}(1-\alpha)((1-p_{t<9}) + p_{t<9}(1-\alpha)) + \\ & (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) - (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \end{aligned} \tag{51}$$

Using the previously added term, we can simplify the equation to:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + (1-p_{t<9})p_{9\leq t<18}(1-\alpha)((1-p_{t<9}) + p_{t<9}) - \\ & (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \end{aligned} \tag{52}$$

Now, we will group together some terms:

$$\begin{aligned} Honest = {} & (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-p_{9\leq t<18})(1-\alpha)) + (1-p_{t<9})p_{9\leq t<18}(1-\alpha) - \\ & (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \end{aligned} \tag{53}$$

By grouping additional terms, we get:

$$Honest = (p_{t<9}(1-\alpha) + (1-p_{t<9})(1-\alpha)) - (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \tag{54}$$

Finally:

$$Honest = (1-\alpha) - (1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \tag{55}$$

□

In order to prove Theorem 2, we will need to calculate the expected block number for each of the parties under the setting where there is not attack.

Corollary 3. *The only difference between our attack and the normal scenario is that in the latter the honest miner earns an additional block when starting at the* Attack *state and the event* Attacker $t < 9$ *occurs.*

Recall that in our MDP we postponed counting an honest block when transitioning to the Attack *state, thus whenever we encounter an event which returns us to the* Normal *state, we count an additional block towards the honest miner:*

$$HonestNoAttack = P\left(Attack\right) \cdot p_{t<9}\alpha + Honest \tag{56}$$

Indeed, when the above is substituted into the honest miner's share of the rewards, and when the difference between the honest and attack scenarios is taken care of, one can see that the standard fair share is obtained:

$$\frac{AttackerNoAttack}{AttackerNoAttack + HonestNoAttack} = \alpha \tag{57}$$

We will now prove Theorem 2.

Theorem 2. *Let there be some block b_0. If the attacker uses the RUM mining strategy, its expected relative share of main-chain blocks will be larger than mining honestly, while the absolute number will remain the same.*

Proof. The fair share is defined as α, while in our attack the actual share is:

$$\frac{Attacker}{Attacker + Honest} \tag{58}$$

We're interested in the gains that the attack gives over the honest scenario. In order to find the improvement in main-chain block-share the attacker gains by adopting the RUM attack, we will calculate the ratio between its share when executing the attack and its share when mining honestly:

$$\frac{Attacker}{Attacker + Honest} \cdot \frac{AttackerNoAttack + HonestNoAttack}{AttackerNoAttack} \tag{59}$$

By substituting the right multiplicand with the fair share, we get:

$$\frac{Attacker}{Attacker + Honest} \cdot \frac{1}{\alpha} \tag{60}$$

Now, by using Claim 6:

$$\frac{\alpha}{\alpha + (1-\alpha) - (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)} \cdot \frac{1}{\alpha} \tag{61}$$

We will now simplify this to:

$$\frac{1}{\alpha + (1-\alpha) - (1 - p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)} \tag{62}$$

Slightly rearranging:

$$\frac{1}{1 - \alpha p_{t<9}p_{9\leq t<18}\left(1 - p_{t<9}\right)\left(1-\alpha\right)} \tag{63}$$

By definition, $\alpha \in (0,1)$ and $p_{t<9}, p_{9\leq t<18} \in (0,1)$, thus:

$$\alpha p_{t<9}p_{9\leq t<18}\left(1 - p_{t<9}\right)\left(1-\alpha\right) \in (0,1) \tag{64}$$

And similarly:

$$1 - \alpha p_{t<9}p_{9\leq t<18}\left(1 - p_{t<9}\right)\left(1-\alpha\right) \in (0,1) \tag{65}$$

Finally:

$$\frac{Attacker}{Attacker + Honest} \cdot \frac{1}{\alpha} = \frac{1}{1 - \alpha p_{t<9} p_{9 \le t < 18} (1 - p_{t<9})(1 - \alpha)} > 1 \tag{66}$$

So, we can conclude that indeed the yields a larger share of main-chain blocks than mining honestly. □

D.5 Expected ETH Share

We will now show that the attacker's expected share of the ETH is larger than its fair share.

Theorem 5. *Let there be some block b_0. If the attacker uses RUM, its share of expected profits will be larger than mining honestly.*

To prove this, we will solve the MDP. First note that the MDP is identical, so as shown in Claim 4, the stationary distribution is:

$$P(Attack) = (1 - p_{t<9}) p_{9 \le t < 18} (1 - \alpha) P(Normal) \qquad = 1 - P(Attack)$$

Claim 7. *The attacker's expected received ETH is:*

$$Attacker = \alpha(R + F)$$

Proof. The attacker mines a single block in each of the following transitions:

(1) Started at the *Normal* state, and the event *Attacker at any time* occurred.
(2) Started at the *Attack* state, and any of the events *Attacker* $t < 9$, *Attacker* $t \ge 9$ occurred.

In both cases it receives $R + F$ ETH.

Thus its expected received ETH is:

$$Attacker = P(Normal) \cdot \alpha \cdot (R + F) + P(Attack) \cdot (p_{t<9}\alpha + (1 - p_{t<9})\alpha) \cdot (R + F) \tag{67}$$

Using algebra and substituting for the state probabilities we get:

$$Attacker = \alpha(R + F) \tag{68}$$

□

We will now do the same for the honest miner. In this analysis we assume the best case for the honest miner, in which it is immediately added as an uncle, receiving $\frac{7}{8}R$ ETH as a reward, and no fees. This is the *worst* case for the attacker, as it decreases the attacker's share.

Claim 8. *The honest miner's expected received ETH is:*

$$Honest = (R + F)\left((1 - \alpha) - \frac{1}{8}(1 - p_{t<9}) p_{9 \le t < 18} (1 - \alpha)(p_{t<9}\alpha)\right)$$

Proof. It mines two blocks when starting at the *Attack* state and the event *Honest* $t < 9$ *or* $18 \le t$ occurred, and mines a single block at each of the following transitions:

(1) When starting at the *Normal* state and either *Honest* $t < 9$ or *Honest* $t \ge 18$ occurred.
(2) When starting at the *Attack* state and *Honest* $9 \le t < 18$ occurred.

In all of those cases, it receives $R + F$ per mined block. On the other hand, in the event that the adversary succeeds in its attack, i.e. starting in the *Attack* state and *Attacker* $t < 9$ has occurred, it receives at most $\frac{7}{8}R$ ETH.

After tallying up all events, we arrive at the following expected received ETH:

$$\begin{aligned} Honest = (R+F)\cdot\Big(&P\left(Normal\right)\cdot p_{t<9}\left(1-\alpha\right)+P\left(Normal\right)\cdot\left(1-p_{t<9}\right)\left(1-p_{9\leq t<18}\right)\left(1-\alpha\right)+\\ &P\left(Attack\right)\cdot 2\left(1-\alpha\right)\left(p_{t<9}+\left(1-p_{t<9}\right)\left(1-p_{9\leq t<18}\right)\right)+\\ &P\left(Attack\right)\left(\left(1-p_{t<9}\right)\alpha+\left(1-p_{t<9}\right)p_{9\leq t<18}\left(1-\alpha\right)\right)\Big)+\frac{7}{8}(R)\cdot\Big(P(Attack)(p_{t<9}\alpha)\Big) \end{aligned} \tag{69}$$

Notice that the terms multiplied by $(R+F)$ are exactly the ones analyzed in Claim 6. Therefore, following the exact same calculation and plugging in the remaining $P(Attack)$ term, we get:

$$\begin{aligned} Honest = (R+F)\Big((1-\alpha)-(1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)+\\ \frac{7}{8}(R)((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)) \end{aligned} \tag{70}$$

Combining the terms we get:

$$Honest = (R+F)(1-\alpha)-(\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big) \tag{71}$$

□

Corollary 4. *Similarly to Corollary 3, the only difference between our attack and the normal scenario is that in the latter the honest miner earns an additional $\frac{1}{8}R$ for mining a main-chain block as well as F in fees when starting at the* Attack *state and the event* Attacker $t < 9$ *occurs.*

Recall that in our MDP we postponed counting an honest block when transitioning to the Attack *state, thus whenever we encounter an event which returns us to the* Normal *state, we count an additional block towards the honest miner:*

$$HonestNoAttack = P\left(Attack\right)\cdot p_{t<9}\alpha(\frac{1}{8}R+F)+Honest \tag{72}$$

Indeed, when the above is substituted into the honest miner's share of the rewards, and when the difference between the honest and attack scenarios is taken care of, one can see that the standard fair share is obtained:

$$\frac{AttackerNoAttack}{AttackerNoAttack+HonestNoAttack}=\alpha \tag{73}$$

We will now prove the theorem.

Proof. The fair share is defined as α, while in our attack the actual share is:

$$\frac{Attacker}{Attacker+Honest} \tag{74}$$

We're interested in the gains that the attack gives over the honest scenario. In order to find the improvement in main-chain block-share the attacker gains by adopting the RUM attack, we will calculate the ratio between its share when executing the attack and its share when mining honestly:

$$\frac{Attacker}{Attacker+Honest}\cdot\frac{AttackerNoAttack+HonestNoAttack}{AttackerNoAttack} \tag{75}$$

$$\frac{Attacker}{Attacker+Honest}\cdot\frac{1}{\alpha} \tag{76}$$

Using Claim 8, we first compute $Attacker + Honest$:

$$\begin{aligned} Attacker + Honest &= (R+F)\alpha + (R+F)(1-\alpha) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big) \\ &= (R+F) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big) \end{aligned} \tag{77}$$

Plugging that value into the previous equation we get:

$$\frac{\alpha(R+F)}{(R+F) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)} \cdot \frac{1}{\alpha} \tag{78}$$

We will now simplify this to:

$$\begin{aligned} &\frac{(R+F)}{(R+F) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\ \leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)} \\ &= 1 + \frac{(\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)}{(R+F) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)} \end{aligned} \tag{79}$$

As noted previously, $\alpha \in (0,1)$ and $p_{t<9}, p_{9\leq t<18} \in (0,1)$, so:

$$(1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha) \in (0,1) \tag{80}$$

In addition, $R, F > 0$, so the numerator in the above term is positive. Furthermore:

$$\begin{aligned} (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\ \leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big) &< \frac{1}{8}R+F \\ &< R+F \end{aligned} \tag{81}$$

So the denominator is positive as well. This means that:

$$\begin{aligned} c &= \frac{(\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)}{(R+F) - (\frac{1}{8}R+F)\Big((1-p_{t<9})p_{9\leq t<18}(1-\alpha)(p_{t<9}\alpha)\Big)} \\ &> 0 \end{aligned} \tag{82}$$

The following inequality holds:

$$\frac{Attacker}{Attacker + Honest} \cdot \frac{1}{\alpha} = 1 + c > 1 \tag{83}$$

So, we can conclude that indeed the attack is more profitable than mining honestly. □

E ADDITIONAL RELATED WORK

In this section we will go over some additional related papers which were not mentioned in Section 9.

Increasing Mining Profits. In our paper, we have presented an attack which allows Ethereum miners to increase their relative share of the profits beyond their actual relative share of the hash-power, in spite of Ethereum's purported goal of giving miners rewards in a "fair" manner, e.g. according to their hash-rate.

Other works have attempted to do this, too. While not an attack, the ***AsicBoost*** mining technique allows faster Bitcoin mining using hardware optimizations tailored specifically for Bitcoin mining hardware [64]. Such hardware is also called an application specific integrated circuit (ASIC) [128]. AsicBoost increases mining profits in a riskless manner too, but it relies on special hardware to do so.

Increasing the Block-rate. In our work, we manipulate timestamps in order to increase mining difficulty and thus obtain blocks which always win ties. The opposite manipulation, modifying timestamps to *decrease* difficulty, is also known as the ***Timewarp*** attack, and has been suggested by [55] not as an attack, but as a possible method to increase Bitcoin's block-rate without performing major protocol modifications. The lower difficulty precludes the possibility of using this attack to replace existing blocks and making a profit.

Similarly, the ***Stretch & Squeeze*** attack attempted to increase and decrease the block-rate in both Bitcoin and Ethereum, but without manipulating timestamps. This work has already been covered in Section 9.

Other Forms of Attacks. Some works have attempted to find evidence of other kinds of attacks, too. For example, block-withholding attacks which are performed by mining pool members, against the mining pool itself [77, 99]. This entails sharing valid mining shares which do not pass the block validity threshold and withholding shares that do. This attack reduces the attacked pool's profits while still maintaining the attacker's profits for the shares that were revealed. Such attacks have been seen in the wild in Bitcoin [21].

Our work has focused on attacking Ethereum's underlying PoW layer. Notably, miners have at their disposal other means to either attack the consensus mechanism or earn excessive profits.

For example, [129] provides a fuzzing framework that can create valid Ethereum transactions which break the consensus in various ways, for example by causing targeted miners to fork the block-DAG.

A strand of works have formulated exploits which are broadly termed miner-extractable value (MEV) exploits. These allow miners to profit by manipulating the application layer, e.g. the smart-contracts running on top of Ethereum's blockchain [17]. Malicious transaction-ordering, front-running and back-running transactions are examples of such attacks, and have been observed in Ethereum [22, 87, 93, 95, 96]. Such exploits are outside the scope of our paper.

Attacks on Smaller Cryptocurrencies. PoW-consensus was designed to prevent *Double Spending* attacks from being performed by an attacker controlling a majoring of the hash-power active on the network. Such attacks broadly follow this recipe: an attacker broadcasts a transaction in return for some good (e.g. the transaction was used by the attacker to buy coffee), while mining a secret fork of the blockchain, starting from the last block which did not include the transmitted transaction. Then, after receiving goods for which the transaction paid, the attacker reveals the secret fork, if it indeed contains enough work to be considered as the main-chain [89, 111].

For example, in Bitcoin the rule-of-thumb is that one should wait until a transaction is at least 6 blocks deep before accepting the transaction, as classic analysis shows that at such depth it is improbable for an attacker with less than 50% of the hash-rate to succeed in forking the chain at a block which is an ancestor of the block that included the transaction.

As can be seen by this attack, the security of cryptocurrencies heavily depends on the amount of honest hash-rate actively mining them, and cryptocurrencies with less active hash-power can be more easily attacked.

Ethereum Classic is a considerably less popular fork of Ethereum, both according to the total active hash-power and when considering economic metrics such as the exchange-rate between the two and the number of transactions [71]. As the two cryptocurrencies share the same consensus mechanism and rely on the same hash-function, even small Ethereum miners can use their hash-power to perform a majority attack on Ethereum Classic, and [63] covers a suspected case of such an attack.

F REPRODUCIBILITY

All code and data used in this paper are available in the following Google Drive link:

https://drive.google.com/drive/folders/12MupjY-DRWMDDQ4X6RwK01v-HWyORAkT

To maintain anonymity, the account in which these files are stored was opened under the pseudonym *Uncle Maker*.

In order for our work to be fully reproducible, we have packaged our code as a standalone Python package which obtains all required data from a full Ethereum node, and then produces all the figures used in the paper.

As extracting and processing all this data requires considerable time and resources, the aforementioned link contains compressed copies of the processed data.

F.1 Data

All the data used in our paper is included in the *./data/* folder, available in the previous Google Drive link. Almost all of this data can be obtained by running the code we provide in Appendix F.8 on a *full* Ethereum node; such nodes retain all main-chain blocks, starting from the genesis [40]. In addition, we have made our data extraction code available as an open-source python library [126]. This library communicates with an Ethereum node, such as geth, using the standard Ethereum RPC interface.

The only data which was not obtained using this method is *miners.csv*, which contains a list of mining pool addresses and the name of the mining pool they are associated with. This was obtained from [44], and is not inherent to our work (besides giving us the possibility to point fingers at F2Pool and other misbehaving miners).

F.2 Alternative Sources of Data

Although Ethereum main-chain data is widely available, it is not necessarily easily accessible, with a full synchronization of a performance-oriented client such as Erigon requiring at least 3 days and a solid state drive (SSD) with 3TB of free space [40]. To that end, we would like to suggest multiple alternative data sources:

Free-Access Nodes. archivenode.io provides free access to an Ethereum archive node, but requires applying for the service, with applications going through a review process. infura.io provides limited free access which requires registration too.

Block Distribution Times and Timestamps. Block distribution times and timestamps are available on [6, 10, 48]. These sources affirm works such as [57, 72], which claim that the latency between miners is extremely small. Specifically, [72], the more recent one, gathered data which showed that 85% of blocks propagate in less than 100 milliseconds.

Transaction Fees. Fees are available on [10, 45–47]. Etherscan's daily transaction fees were saved in *transaction_fees.csv*, and the number of blocks mined per day in *block_count_rewards.csv*. Both can be used to assess the average amount of transaction fees earned per-block.

F.3 Running Our Code

We will now go over the installation and usage instructions for the code we used when writing this paper, and the hardware which we used to run it.

Software. All the code used for the paper is contained in the *./code/* folder, and can be executed in the following manner:

(1) Download and install Python 3.9.12.
(2) Open a command-line shell, and change the current directory to this paper's *code* directory.
(3) Our code requires *Python 3.9.12* [118], *matplotlib 3.5.2*, *numpy 1.22.4* [65], *scipy 1.8.1* [119], *pandas 1.4.2* [85] and *sagemath 9.6* [116]. To install all prerequisites, run:
`python setup.py install`
(4) Given a full Ethereum node can be accessed using the uniform resource locator (URL) *<FullEthereumNodeURL>*, the code can be executed following command:
`python main.py <FullEthereumNodeURL>`

Hardware. All graphs used in the paper were generated using the aforementioned code on a laptop equipped with an *Intel Core i7-11370H* central processing unit (CPU) and *64GB* of random-access memory (RAM), running *Windows 11*.

F.4 Go-Ethereum Patch

Go-Ethereum, also called *geth*, is the most popular Ethereum client in use today [38, 40, 43, 73].

We will now describe how to modify the current time for a geth instance running in a Docker container [25]. This allows miners to mine blocks with timestamps which are not taken from their system's clock. Surprisingly, this was not an easy task.

Modifying the local clock for a UNIX-like operating system can usually be achieved using the *date* command. Unfortunately, this approach changes the local time for all Docker containers, even if they use different images.

Luckily, there are two software packages, *libfaketime* and *datefudge* which allow setting different local times for each Docker instance. But, this approach does not work with Go's *time.Now()* function, because Go is a statically-linked language, and these packages rely on changing dynamic links at runtime. This is in spite of some discussions in the Ethereum community, which hint that miners did use libfaketime in the past, specifically a user called Cody Burns (with username @realcodywburns) posted on September 17th, 2017 at 19:30 [41]: *The dirty secret in mining is that most use libfaketime to avoid uncles. Miners are alowed to report their own block time, and so long as it is within reasonable tolerance.*

Thus, we attempted using various Go packages which allow replacing Go functions by "monkey-patching" these functions, and replacing their bodies with calls to the functions which replace them. This, too, has failed, as currently-available monkey-patching methods in Go are not thread-safe, leading to runtime errors.

Finally, we modified Go's standard library, specifically the *time.go* file. Additionally, we opted to use *libfaketime* just in case. We provide a *Dockerfile* that performs all these modifications automatically.

In order to install our patch, follow these instructions:

(1) Download, install and run Docker [11]. Detailed instructions are provided on their web-site [25].
(2) In some path *<path>*, create a file called *Dockerfile*. The file's contents should be identical to Appendix F.6.
(3) Open a command-line shell, and change the current directory to *<path>*.
(4) Run the following command to create a Docker image which holds a version of geth which was patched to execute the attack:
`docker build -t unclemaker/geth .`
(5) Use the following command to run the patched version of geth:
`run --rm -it unclemaker/geth`

F.5 Ethereum's Documentation and Code

Ethereum has changed considerably since its inception, as evident by the Ethereum improvement proposals (EIPs) which have been approved over its lifetime [37], with some changes in its DAA [14].

Specifically, Ethereum's white paper hasn't been updated with some of these changes, with Ethereum's official documentation advising users to follow their yellow paper instead [15, 125]. Additionally, it is well known that sometimes, the actual implementation of cryptocurrencies might differ from the protocol specifications given in their documentations.

In the context of our paper, we have found no such discrepancies. Still, throughout this paper we referred to both the yellow paper and geth's source code, wherever relevant. In both cases, we used permanent links to the most recent versions at the time of writing to make sure that even if changes are made after the publication of this paper, our links will still point to the versions which our paper relied on.

GETH PATCHES

F.6 Geth Patch to Execute the Preemptive Uncle Maker Attack

```
# To build this docker image:
#   - Do: install docker, see: https://www.docker.com/get-started/
#   - Do: open a terminal
#   - Do: change the current directory to the directory of this Dockerfile
#   - Run: docker build -t unclemaker/geth .
# Now, you can run it like so:
#   - Run: docker run --rm -it unclemaker/geth

# Support setting various labels on the final image
ARG COMMIT=""
ARG VERSION=""
ARG BUILDNUM=""

# Build Geth in a stock Go builder container
FROM golang:1.18-alpine as builder

# Install some prerequisites
RUN apk add --no-cache gcc musl-dev linux-headers git

# Start: libfaketime
```

```
RUN apk add --no-cache make
RUN git clone https://github.com/wolfcw/libfaketime.git /libfaketime
RUN cd /libfaketime && make
# End
# Start: modifying Golang's time library
RUN sed -i 」
↪ 's/sec += unixToInternal - minWall/sec += -1 + unixToInternal - minWall/g'」
↪
↪ /usr/local/go/src/time/time.go
RUN go clean --cache
# End

# Start: geth
RUN git clone https://github.com/ethereum/go-ethereum.git /go-ethereum
RUN cd /go-ethereum && go run build/ci.go install ./cmd/geth
# End

# Pull Geth into a second stage deploy alpine container
FROM alpine:latest

RUN apk add --no-cache ca-certificates
COPY --from=builder /go-ethereum/build/bin/geth /usr/local/bin/

# Start: libfaketime
# The FAKETIME parameter sets the time delta: FAKETIME="-1"
# The FAKETIME_DONT_RESET parameter makes sure time will increment for new
# processes, instead of each new process reverting back to the originally
# set time (see example 4c in wolfcw/libfaketime for more info).
# The LD_PRELOAD parameter sets the location of the library.
ENV FAKETIME="-1"
ENV FAKETIME_DONT_RESET=1
ENV LD_PRELOAD=/usr/local/lib/libfaketime.so.1
COPY --from=builder /libfaketime/src/libfaketime.so.1 /usr/local/lib
# End

# Start: geth
EXPOSE 8545 8546 30303 30303/udp
ENTRYPOINT ["geth"]
# End

# Add some metadata labels to help programatic image consumption
ARG COMMIT=""
ARG VERSION=""
ARG BUILDNUM=""

LABEL commit="$COMMIT" version="$VERSION" buildnum="$BUILDNUM"
```

F.7 Geth Patch to Mitigate Uncle Maker Attacks

```
# To build this docker image:
#   - Do: install docker, see: https://www.docker.com/get-started/
#   - Do: open a terminal
#   - Do: change the current directory to the directory of this Dockerfile
#   - Run: docker build -t unclemaker/geth .
# Now, you can run it like so:
#   - Run: docker run --rm -it unclemaker/geth

# Support setting various labels on the final image
ARG COMMIT=""
ARG VERSION=""
ARG BUILDNUM=""

# Build Geth in a stock Go builder container
FROM golang:1.18-alpine as builder

# Install some prerequisites
RUN apk add --no-cache gcc musl-dev linux-headers git
RUN apk add sd
 ↪  --repository=http://dl-cdn.alpinelinux.org/alpine/edge/testing/

# Start: geth
RUN git clone https://github.com/ethereum/go-ethereum.git /go-ethereum
# The following line fixes the 3 cases
RUN sd 'reorg := externTd.Cmp(localTD) > 0' ⌋
 ↪  'if header.UncleHash == types.EmptyUncleHash && header.ParentHash == current.Parent
 ↪
 ↪  /go-ethereum/core/forkchoice.go
# The following line fixes the min. diff issue
RUN sd 'MinimumDifficulty      = big.NewInt(131072)'
 ↪  'MinimumDifficulty      = big.NewInt(4194304)'
 ↪  /go-ethereum/params/protocol_params.go
RUN cd /go-ethereum && go run build/ci.go install ./cmd/geth
# End

# Pull Geth into a second stage deploy alpine container
FROM alpine:latest

RUN apk add --no-cache ca-certificates
COPY --from=builder /go-ethereum/build/bin/geth /usr/local/bin/

# Start: geth
EXPOSE 8545 8546 30303 30303/udp
ENTRYPOINT ["geth"]
# End

```

```
# Add some metadata labels to help programatic image consumption
ARG COMMIT=""
ARG VERSION=""
ARG BUILDNUM=""

LABEL commit="$COMMIT" version="$VERSION" buildnum="$BUILDNUM"
```

DATA EXTRACTION AND GRAPHS

F.8 Code Requirements

```
argon2-cffi==21.3.0
argon2-cffi-bindings==21.2.0
asttokens==2.0.5
attrs==21.4.0
backcall==0.2.0
beautifulsoup4==4.11.1
bleach==5.0.0
certifi==2022.5.18.1
cffi==1.15.0
charset-normalizer==2.0.12
colorama==0.4.4
cycler==0.11.0
debugpy==1.6.0
decorator==5.1.1
defusedxml==0.7.1
entrypoints==0.4
executing==0.8.3
fastjsonschema==2.15.3
fonttools==4.33.3
idna==3.3
ipykernel==6.13.1
ipython==8.4.0
ipython-genutils==0.2.0
ipywidgets==7.7.0
jedi==0.18.1
Jinja2==3.1.2
jsonschema==4.6.0
jupyter==1.0.0
jupyter-client==7.3.4
jupyter-console==6.4.3
jupyter-core==4.10.0
jupyterlab-pygments==0.2.2
jupyterlab-widgets==1.1.0
kiwisolver==1.4.2
MarkupSafe==2.1.1
matplotlib==3.5.2
matplotlib-inline==0.1.3
mistune==0.8.4
```

```
mpmath==1.2.1
nbclient==0.6.4
nbconvert==6.5.0
nbformat==5.4.0
nest-asyncio==1.5.5
notebook==6.4.12
numpy==1.22.4
packaging==21.3
pandas==1.4.2
pandocfilters==1.5.0
parso==0.8.3
pickleshare==0.7.5
Pillow==9.1.1
prometheus-client==0.14.1
prompt-toolkit==3.0.29
psutil==5.9.1
pure-eval==0.2.2
pycparser==2.21
Pygments==2.12.0
pyparsing==3.0.9
pyrsistent==0.18.1
python-dateutil==2.8.2
pytz==2022.1
pywin32==304
pywinpty==2.0.5
pyzmq==23.1.0
qtconsole==5.3.1
QtPy==2.1.0
requests==2.27.1
scipy==1.8.1
seaborn==0.11.2
Send2Trash==1.8.0
six==1.16.0
soupsieve==2.3.2.post1
stack-data==0.2.0
sympy==1.10.1
terminado==0.15.0
tinycss2==1.1.1
tornado==6.1
traitlets==5.2.2.post1
urllib3==1.26.9
wcwidth==0.2.5
webencodings==0.5.1
widgetsnbextension==3.6.0
```

F.9 setup.py

```
from setuptools import setup

```

```

def readme():
    with open("README.md") as f:
        return f.read()


with open("requirements.txt") as f:
    requirements = f.read().splitlines()

setup(
    name="unclemaker",
    version="0.0.1",
    description="All code used in the UncleMaker paper.",
    url="https://github.com/UncleMaker/UncleMaker",
    author="Uncle Maker Authors",
    packages=["unclemaker"],
    long_description=readme(),
    python_requires=">=3.9",
    install_requires=requirements,
)
```

F.10 main.py

```
"""The main file for the Uncle Maker paper.
"""
import sys
import obtain_data
import paper_stuff
import generate_graphs

if __name__ == "__main__":
    arguments = sys.argv[1:]
    if len(arguments) != 1:
        print("Usage: python main.py <FullEthereumNodeURL>")
        exit()

    mainchain_bodies, uncle_bodies = obtain_data.request_from_node(
        sys.argv[1:].pop(), 15226042
    )
    mainchain_bodies_df, uncle_bodies_df = obtain_data.process_data(
        mainchain_bodies, uncle_bodies, "miners.csv"
    )
    obtain_data.save(mainchain_bodies_df, uncle_bodies_df)

    paper_stuff.find_f2pool_uncle_makers(mainchain_bodies_df,
    ↪  uncle_bodies_df)
    paper_stuff.verify_mdp()
    generate_graphs.generate()
```

F.11 obtain_data.py

```python
import pandas as pd
import requests
import json
from typing import Union

MainchainBlocks = dict[int, dict[str, Union[int, str]]]
UncleBlocks = dict[str, dict[str, dict[str, Union[int, str]]]]


def request_from_node(url: str, start_from: int) -> tuple[MainchainBlocks, UncleBlocks]:
    """Obtains relevant data from the given full Ethereum node.

    :param url: the full node's URL.
    :type url: str
    :param start_from: the block to start obtaining data from.
    :type start_from: int
    :return: the timestamps and miners of main and uncle blocks.
    :rtype: tuple[dict[int, dict[str, Union[int, str]]], dict[str, dict[str, dict[s
    """
    mainchain_bodies: MainchainBlocks = {}
    uncle_bodies: UncleBlocks = {}
    session = requests.Session()
    headers = {"Content-type": "application/json"}

    # Create multiple batch requests to get the relevant blockchain data
    batch_size: int = 7 * 6466  # Week's worth of blocks
    batches: int = 4 * 12 * 7  # Get the last seven years' worth of data
    for i in range(batches):
        print(start_from, int(100 * i / batches))
        if start_from <= 0:
            break
        uncle_count = {}
        response = session.post(
            url,
            headers=headers,
            data=json.dumps(
                [
                    {
                        "jsonrpc": "2.0",
                        "method": "eth_getBlockByNumber",
                        "params": [hex(idx), False],
                        "id": idx,
                    }
                    for idx in range(
```

```
                        max(start_from - batch_size, 0), max(start_from, 0)
                    )
                ]
            ),
        )
        for cur_block in response.json():
            # To save space, only keep timestamps, miner addresses, and uncles
            mainchain_bodies[int(cur_block["result"]["number"], 16)] = {
                "timestamp": int(cur_block["result"]["timestamp"], 16),
                "miner": cur_block["result"]["miner"],
                "uncles": cur_block["result"]["uncles"],
            }
            if cur_block["result"]["uncles"] != []:
                uncle_count[cur_block["id"]] =
                ↪  len(cur_block["result"]["uncles"])

        # Get uncles, if count > 0
        if len(uncle_count) == 0:
            continue
        response = session.post(
            url,
            headers=headers,
            data=json.dumps(
                [
                    {
                        "jsonrpc": "2.0",
                        "method": "eth_getUncleByBlockNumberAndIndex",
                        "params": [hex(block_num), hex(idx)],
                        "id": block_num,
                    }
                    for block_num, count in uncle_count.items()
                    for idx in range(0, count)
                ]
            ),
        )
        for cur_block in response.json():
            cur_num = int(cur_block["result"]["number"], 16)
            uncle_bodies[cur_block["result"]["hash"]] = {
                "number": cur_num,
                "timestamp": int(cur_block["result"]["timestamp"], 16),
                "miner": cur_block["result"]["miner"],
            }

        start_from -= batch_size
    return mainchain_bodies, uncle_bodies


```

```python
def process_data(
    mainchain_bodies: MainchainBlocks, uncle_bodies: UncleBlocks, miner_names_file: str
):
    """Processes the inputs and converts them to pandas DataFrames.

    :return: the processed DataFrames.
    """
    # This assumes that all files should be read from and saved to '../data/'
    # Load data
    miner_names = {
        row["address"]: row["name"]
        for _, row in pd.read_csv(f"../data/{miner_names_file}").iterrows()
    }

    # Convert to pandas DFs, sort by timestamps for ease-of-use
    mainchain_bodies_df = pd.DataFrame.from_dict(
        mainchain_bodies, orient="index", columns=["timestamp", "miner", "uncles"]
    ).sort_index()
    uncle_bodies_df = pd.DataFrame.from_dict(
       uncle_bodies, orient="index", columns=["number", "timestamp", "miner"]
    ).sort_values(["number", "timestamp"])

    # Associate blocks with siblings
    uncle_count = uncle_bodies_df.value_counts("number")
    mainchain_bodies_df["sibling_number"] = mainchain_bodies_df.apply(
        lambda row: uncle_count.get(row.name, 0), axis=1
    )

    # Associate blocks with miners and time differences
    mainchain_bodies_df["miner_name"] = mainchain_bodies_df["miner"].apply(
        lambda key: miner_names.get(key, key[:4] + "..." + key[-2:])
    )
    uncle_bodies_df["miner_name"] = uncle_bodies_df["miner"].apply(
        lambda key: miner_names.get(key, key[:4] + "..." + key[-2:])
    )
    mainchain_bodies_df["timediff"] = mainchain_bodies_df[
        "timestamp"
    ] - mainchain_bodies_df["timestamp"].shift(1)
    uncle_bodies_df["timediff_sibling"] = uncle_bodies_df.apply(
        lambda row: row["timestamp"]
        - mainchain_bodies_df["timestamp"].loc[row["number"]],
        axis=1,
    )
    uncle_bodies_df["timediff_parent"] = uncle_bodies_df.apply(
        lambda row: row["timestamp"]
```

```
        - mainchain_bodies_df["timestamp"].loc[row["number"] - 1],
        axis=1,
    )

    return mainchain_bodies_df, uncle_bodies_df


def save(mainchain_bodies_df, uncle_bodies_df) -> None:
    """Saves the inputs to '../data/'.
    """
    mainchain_bodies_df.to_pickle(
        f"../data/mainchain_{mainchain_bodies_df.index[0]}_{mainchain_bodies
        ↪  _df.index[-1]}.pickle",
        compression="bz2",
    )
    uncle_bodies_df.to_pickle(
        f"../data/uncle_{uncle_bodies_df.iloc[0]['number']}_{uncle_bodies_df
        ↪  .iloc[-1]['number']}.pickle",
        compression="bz2",
    )
```

F.12 generate_graphs.py

```
import matplotlib.ticker as mtick
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np


def generate():
    # Avoid type 3 fonts in pdf figures
    plt.rcParams["pdf.fonttype"] = 42
    plt.rcParams["ps.fonttype"] = 42

    # Seaborn styling
    plt.style.use("seaborn-deep")
    sns.set_style("whitegrid")
    palette = "hls"

    # Load data
    mainchain_bodies_df = pd.read_pickle(
        "../data/mainchain_0_14816570.pickle", compression="bz2"
    )
    uncle_bodies_df = pd.read_pickle(
        "../data/uncle_1_14816569.pickle", compression="bz2"
    )
```

```
26
27      # Limit the blocks we're looking at
28      start_block: int = 12 * (10 ** 6)
29      timediff_limit = 18
30
31      selected_blocks = mainchain_bodies_df[
32          (mainchain_bodies_df.index >= start_block)
33          & (mainchain_bodies_df["timediff"] <= timediff_limit)
34      ]
35      selected_uncles = uncle_bodies_df[uncle_bodies_df["number"] >=
        ↪  start_block]
36
37      # Calculate timestamp differences
38      # Note: the genesis block and its children distort everything because the
39      # genesis has a timestamp of 0, so better to start afterwards
40      print(
41          f"All data is since block {start_block}.\n",
42          ⌋
            ↪  "Mean and median timestamp diffs between mainchain blocks and parents:"⌋
            ↪  ,
43          np.mean(mainchain_bodies_df[start_block:]["timediff"]),
44          np.median(mainchain_bodies_df[start_block:]["timediff"]),
45          "\n",
46          "Mean and median diffs between uncles and parents:",
47          np.mean(selected_uncles["timediff_parent"]),
48          np.median(selected_uncles["timediff_parent"]),
49          "\n",
50          "Mean and median diffs between uncles and siblings:",
51          np.mean(selected_uncles["timediff_sibling"]),
52          np.median(selected_uncles["timediff_sibling"]),
53      )
54
55      # Associate timestamp differences with miners
56      counts = pd.DataFrame(
57          data=0,
58          columns=range(1, timediff_limit + 1),
59          index=set(selected_blocks["miner_name"]),
60      )
61      for (miner, tdiff), val in selected_blocks.value_counts(
62          ["miner_name", "timediff"]
63      ).iteritems():
64          counts.loc[miner][tdiff] = val
65      counts = counts.iloc[counts.sum(axis=1).argsort()[::-1]]  # Sort by total
        ↪  blocks
66      top4 = counts[:4]
67
68      # Plot a histogram of blocktimes for all miners
```

```
    all_miners = counts.sum(axis=0)
    p = sns.barplot(
        x=all_miners.index, y=all_miners, color=sns.color_palette(palette,
        ↪  100)[62],
    )
    p.set_xlabel("Timestamp difference from parent, in seconds")
    p.set_ylabel("Number of blocks")
    plt.savefig(
        "../images/timestampDiffAllMiners.pdf", bbox_inches="tight",
        ↪  pad_inches=0
    )
    plt.show()
    plt.close()

    # Associate uncle timestamp differences with miners
    uncle_counts = pd.DataFrame(
        data=0,
        columns=range(1, timediff_limit + 1),
        index=set(selected_uncles["miner_name"]),
    )
    for (miner, tdiff), val in selected_uncles.value_counts(
        ["miner_name", "timediff_parent"]
    ).iteritems():
        uncle_counts.loc[miner][tdiff] = val
    uncle_counts = uncle_counts.iloc[
        uncle_counts.sum(axis=1).argsort()[::-1]
    ]  # Sort by total blocks
    uncle_top4 = uncle_counts[:4]

    # Plot a histogram of uncletimes for all miners
    all_miners = uncle_counts.sum(axis=0)
    p = sns.barplot(
        x=all_miners.index, y=all_miners, color=sns.color_palette(palette,
        ↪  100)[62],
    )
    p.set_xlabel("Timestamp difference from parent, in seconds")
    p.set_ylabel("Number of uncles")
    plt.savefig(
        "../images/timestampDiffAllMinersUncles.pdf", bbox_inches="tight",
        ↪  pad_inches=0
    )
    plt.show()
    plt.close()

    # Plot a histogram of blocktimes per miner, for the top 5 miners
    p = sns.barplot(
        x="TimestampDiff",
```

```
        y="value",
        hue="index",
        palette=palette,
        data=top4.reset_index().melt(id_vars="index",
        ↪  var_name="TimestampDiff"),
    )
    p.set_xlabel("Timestamp difference from parent, in seconds")
    p.set_ylabel("Number of blocks")
    p.legend(title="Miners", loc="best")
    plt.savefig(
        "../images/timestampDiff12000000.pdf", bbox_inches="tight",
        ↪  pad_inches=0
    )
    plt.show()
    plt.close()

    # Plot a histogram of uncletimes per miner, for the top 5 miners
    p = sns.barplot(
        x="TimestampDiff",
        y="value",
        hue="index",
        palette=palette,
        data=uncle_top4.reset_index().melt(id_vars="index",
        ↪  var_name="TimestampDiff"),
    )
    p.set_xlabel("Timestamp difference from parent, in seconds")
    p.set_ylabel("Number of uncles")
    p.legend(title="Miners", loc="best")
    plt.savefig(
        "../images/timestampDiff12000000Uncles.pdf", bbox_inches="tight",
        ↪  pad_inches=0
    )
    plt.show()
    plt.close()

    # Plot a histogram of blocktimes for F2Pool
    F2Pool = (
        mainchain_bodies_df[
            (mainchain_bodies_df["miner_name"] == "F2Pool")
            & (mainchain_bodies_df.index >= 12000000)
        ]
        .groupby("timediff")
        .count()["miner_name"]
        .append(pd.Series([0 for _ in range(1, 5)], index=[9 * i for i in
        ↪  range(1, 5)]))
        .sort_index()
```

```
153     )  # Note that pandas cannot count 0 occurences of something, so we add
        ↪  them just to make the empty places obvious!
154     _, ax = plt.subplots(figsize=(11.7, 8.27))
155     p = sns.barplot(
156         ax=ax,
157         x=F2Pool.index[:37].astype(int),
158         y=F2Pool.values[:37],
159         color=sns.color_palette(palette, 100)[62],
160     )
161     p.set_xlabel("Timestamp difference from parent, in seconds")
162     p.set_ylabel("Number of blocks mined by F2Pool")
163     plt.savefig("../images/timestampDiff12000000F2Pool.pdf", pad_inches=0)
164     plt.show()
165     plt.close()
166
167     # Find pools that avoid timestamps divisible by 9
168     uncle_makers = counts[
169         np.all(
170             [counts[timediff] == 0 for timediff in counts.columns if timediff
                ↪  % 9 == 0],
171             axis=0,
172         )
173     ]
174
175     # When did the top 5 uncle makers start avoiding such blocks?
176     for miner in uncle_makers.index[:5]:
177         print(
178             f"Miner {miner}
                ↪  stopped mining blocks with a time-diff divisible by 9 since block:"
                ↪  ,
179             mainchain_bodies_df[mainchain_bodies_df["miner_name"] ==
                ↪  miner]["timediff"][
180                 ::-1
181             ]
182             .eq(9)
183             .idxmax()
184             + 1,
185         )
186
187     # Plot a histogram of blocktimes per miner, for the top 4 uncle makers
188     p = sns.barplot(
189         x="TimestampDiff",
190         y="value",
191         hue="index",
192         data=uncle_makers[:4]
193         .reset_index()
194         .melt(id_vars="index", var_name="TimestampDiff"),
```

```
        palette=palette,
    )
    p.set_xlabel("Timestamp difference from parent, in seconds")
    p.set_ylabel("Number of blocks")
    p.legend(title="Miners", loc="best")
    plt.savefig(
        "../images/timestampDiffUncleMakers.pdf", bbox_inches="tight",
        ↪  pad_inches=0
    )
    plt.show()
    plt.close()

    # Pie chart for miner portion, for the top 5 miners
    slice_num = 4
    miner_mainchain_count = counts.sum(axis=1)
    data = list(miner_mainchain_count[:slice_num]) + [
        sum(miner_mainchain_count[slice_num:])
    ]
    labels = list(miner_mainchain_count[:slice_num].index) + ["Others"]
    cur_color = iter(
        sns.color_palette(palette, slice_num + 2)
    )  # Make sure we have enough colors for the next pie chart
    colors = {label: next(cur_color) for label in labels}
    p = plt.pie(
        data,
        labels=labels,
        colors=[colors[label] for label in labels],
        autopct="%.0f%%",
    )
    plt.savefig("../images/mainchainPortion.pdf", bbox_inches="tight",
    ↪  pad_inches=0)
    plt.show()
    plt.close()

    # Pie chart for miner portion, for the top 5 miners
    miner_uncle_count = uncle_bodies_df["miner_name"].value_counts()
    data = list(miner_uncle_count[:slice_num]) +
    ↪  [sum(miner_uncle_count[slice_num:])]
    labels = list(miner_uncle_count[:slice_num].index) + ["Others"]
    colors |= {label: next(cur_color) for label in labels if label not in
    ↪  colors}
    p = plt.pie(
        data,
        labels=labels,
        colors=[colors[label] for label in labels],
        autopct="%.0f%%",
    )
```

```
238     plt.savefig("../images/unclePortion.pdf", bbox_inches="tight",
        ↪  pad_inches=0)
239     plt.show()
240     plt.close()
241
242     # Create date columns, for ease of use
243     mainchain_bodies_df["date"] = pd.DatetimeIndex(
244         pd.to_datetime(mainchain_bodies_df["timestamp"], unit="s")
245     )
246     mainchain_bodies_df["year"] =
        ↪  pd.DatetimeIndex(mainchain_bodies_df["date"]).year
247     mainchain_bodies_df["month"] =
        ↪  pd.DatetimeIndex(mainchain_bodies_df["date"]).month
248     uncle_bodies_df["date"] = pd.DatetimeIndex(
249         pd.to_datetime(uncle_bodies_df["timestamp"], unit="s")
250     )
251     uncle_bodies_df["year"] = pd.DatetimeIndex(uncle_bodies_df["date"]).year
252     uncle_bodies_df["month"] =
        ↪  pd.DatetimeIndex(uncle_bodies_df["date"]).month
253
254     # Create an easy-to-use DF for timestamp diffs which are divisble by 9
255     diffs_divisible_by_nine_per_month = (
256         mainchain_bodies_df[
257             mainchain_bodies_df["miner_name"].isin(
258                 np.concatenate([np.array(top4.index), uncle_makers.index[:5]])
259             )
260             & (mainchain_bodies_df["timediff"] % 9 == 0)
261         ]
262         .groupby(["year", "month", "miner_name"])
263         .count()
264         .sort_values(["year", "month", "miner_name"])
265     )
266     to_drop = diffs_divisible_by_nine_per_month.columns
267     diffs_divisible_by_nine_per_month["count"] =
        ↪  diffs_divisible_by_nine_per_month[
268         "timestamp"
269     ]
270     diffs_divisible_by_nine_per_month.drop(to_drop, axis=1, inplace=True)
271     flat_diffs_divisible =
        ↪  diffs_divisible_by_nine_per_month.index.to_frame(index=False)
272     flat_diffs_divisible["count"] =
        ↪  np.array(diffs_divisible_by_nine_per_month["count"])
273     flat_diffs_divisible["date"] = pd.to_datetime(
274         flat_diffs_divisible["year"].astype(str)
275         + "-"
276         + flat_diffs_divisible["month"].astype(str).map(lambda month:
            ↪  month.zfill(2)),
```

```
        format="%Y-%m",
    )
    flat_diffs_divisible = flat_diffs_divisible.pivot(
        index="date", columns="miner_name", values="count"
    ).fillna(0)

    # Plot a graph of the amount of blocks with timestamp diff. divisible by 9 by date
    fig, ax = plt.subplots()

    plt.ylabel("Amount of blocks with timestamp diff. divisible by 9")
    plt.xlabel("Date")
    ax.xaxis.set_major_formatter(mdates.DateFormatter("%b '%y"))
    ax.xaxis.set_major_locator(mtick.MultipleLocator(160))

    relevant_dates = flat_diffs_divisible[
        (flat_diffs_divisible.index >= "2019-08-01")
        & (flat_diffs_divisible.index <= "2021-09-01")
    ]
    for miner, marker in [("Ethermine", "o"), ("F2Pool", "x"), ("0x99...e3", "|")]:
        plt.plot(
            relevant_dates.index,
            relevant_dates[miner],
            marker=marker,
            markersize=5,
            label=miner,
        )

    ax.legend(loc="best")
    plt.tight_layout()
    plt.savefig(
        "../images/timestampDivisbleBy9OverTime.pdf", bbox_inches="tight", pad_inches=0
    )
    plt.show()
    plt.close()

    # Calculate the monthly "uncle rate" for certain miners (# of uncle / # of main-chain blocks)
    mainchain_monthly_count = (
        mainchain_bodies_df[
            mainchain_bodies_df["miner_name"].isin(
                np.concatenate([np.array(top4.index), uncle_makers.index[:5]])
            )
        ]
        .groupby(["year", "month", "miner_name"])
```

```
        .count()
        .sort_values(["year", "month", "miner_name"])
    )
    to_drop = mainchain_monthly_count.columns
    mainchain_monthly_count["count"] = mainchain_monthly_count["timestamp"]
    mainchain_monthly_count.drop(to_drop, axis=1, inplace=True)
    flat_mainchain_monthly_count =
    ↪  mainchain_monthly_count.index.to_frame(index=False)
    flat_mainchain_monthly_count["count"] =
    ↪  np.array(mainchain_monthly_count["count"])
    flat_mainchain_monthly_count["date"] = pd.to_datetime(
        flat_mainchain_monthly_count["year"].astype(str)
        + "-"
        + flat_mainchain_monthly_count["month"]
        .astype(str)
        .map(lambda month: month.zfill(2)),
        format="%Y-%m",
    )
    flat_mainchain_monthly_count = flat_mainchain_monthly_count.pivot(
        index="date", columns="miner_name", values="count"
    ).fillna(0)

    uncle_monthly_count = (
        uncle_bodies_df[
            uncle_bodies_df["miner_name"].isin(
                np.concatenate([np.array(top4.index), uncle_makers.index[:5]])
            )
        ]
        .groupby(["year", "month", "miner_name"])
        .count()
        .sort_values(["year", "month", "miner_name"])
    )
    to_drop = uncle_monthly_count.columns
    uncle_monthly_count["count"] = uncle_monthly_count["timestamp"]
    uncle_monthly_count.drop(to_drop, axis=1, inplace=True)
    flat_uncle_monthly_count =
    ↪  uncle_monthly_count.index.to_frame(index=False)
    flat_uncle_monthly_count["count"] =
    ↪  np.array(uncle_monthly_count["count"])
    flat_uncle_monthly_count["date"] = pd.to_datetime(
        flat_uncle_monthly_count["year"].astype(str)
        + "-"
        + flat_uncle_monthly_count["month"]
        .astype(str)
        .map(lambda month: month.zfill(2)),
        format="%Y-%m",
    )
```

```
    flat_uncle_monthly_count = flat_uncle_monthly_count.pivot(
        index="date", columns="miner_name", values="count"
    ).fillna(0)

    # Plot a graph of the monthly uncle rate
    fig, ax = plt.subplots()

    plt.ylabel("Monthly uncle rate")
    ax.yaxis.set_major_formatter(mtick.PercentFormatter(decimals=0))

    plt.xlabel("Date")
    ax.xaxis.set_major_formatter(mdates.DateFormatter("%b '%y"))
    ax.xaxis.set_major_locator(mtick.MultipleLocator(160))

    uncle_relevant_dates = flat_uncle_monthly_count.iloc[42:-7]
    mainchain_relevant_dates = flat_mainchain_monthly_count.iloc[42:-7]
    for miner, marker in [("Ethermine", "o"), ("F2Pool", "x"), ("0x99...e3",
    ↪  "|")]:
        uncle_rate = (
            100
            * uncle_relevant_dates[miner]
            / (uncle_relevant_dates[miner] + mainchain_relevant_dates[miner])
        )
        plt.plot(uncle_rate.index, uncle_rate, marker=marker, markersize=5,
        ↪  label=miner)

    ax.legend(loc="best")
    plt.tight_layout()
    plt.savefig("../images/monthlyUncleRate.pdf", bbox_inches="tight",
    ↪  pad_inches=0)
    plt.show()
    plt.close()
```

F.13 paper_stuff.py

```
from sympy import *


def find_f2pool_uncle_makers(mainchain_bodies_df, uncle_bodies_df) -> None:
    """ Find concrete examples of Uncle Maker blocks by F2Pool.
    """
    # The code found a main chain block with number 14772899
    for _, uncle in uncle_bodies_df[::-1].iterrows():
        number = uncle["number"]
        if (
            (uncle["timediff_sibling"] == 1)
            and (mainchain_bodies_df.loc[number]["miner_name"] == "F2Pool")
            and (
```

```
                mainchain_bodies_df.loc[number]["timediff"] // 9
                < uncle["timediff_parent"] // 9
            )
        ):
            print(
                f"""
⌋
↪                   We're looking for a concrete Uncle Maker block mined by F2Pool.
                We're interested in a series of four blocks:
                Grandparent <- parent <- child
                            <- uncle  <-
                Where the parent and uncle have:
                - A timestamp diff. of 1 second,
                - Their timestamps are in diff. 9 sec windows,
                - The parent was mined by F2Pool
                The code found the following example:
                Uncle: {uncle}.
                Grandparent: {mainchain_bodies_df.loc[number - 1]}.
                Parent: {mainchain_bodies_df.loc[number]}.
                Child: {mainchain_bodies_df.loc[number + 1]}.
                """
            )
            break
    print("Couldn't find any Uncle Makers by F2Pool.")


def verify_mdp() -> None:
    """Verify the various MDP calculations done in the paper.
    """
    # Declare variables, 'a' is alpha (the attacker's share)
    a, p0, p1 = symbols("a p0 p1")

    # Stationary distribution
    p_attack = (1 - p0) * p1 * (1 - a)
    p_normal = 1 - p_attack

    # The attacker's expected number of blocks
    attacker = simplify((p_normal * a) + (p_attack * ((p0 * a) + ((1 - p0) *
    ↪  a))))

    # The honest's expected number of blocks
    honest = simplify(
        (p_normal * p0 * (1 - a))
        + (p_normal * (1 - p0) * (1 - p1) * (1 - a))
        + (p_attack * (1 - p0) * a)
        + (p_attack * 2 * (1 - a) * (p0 + ((1 - p0) * (1 - p1))))
        + (p_attack * (1 - p0) * p1 * (1 - a))
```

```
    )
    honest_no_attack = simplify((p_attack * p0 * a) + honest)

    # Attacker's share of the blocks
    share_attack = simplify(attacker / (attacker + honest))
    share_no_attack = simplify(attacker / (attacker + honest_no_attack))

    print(
        f"""
        Expected number of honest blocks, no attack: {honest_no_attack}.
        Expected number of honest blocks, attack: {honest}.
        Expected attacker share, no attack: {share_no_attack}.
        Expected attacker share, attack: {share_attack}.
        Expected attacker improvement: {share_attack / share_no_attack}.
        """
    )
```

G GLOSSARY

All symbols and acronyms used in the paper are summarized in this section.

SYMBOLS

- A Attacker's hash-rate, in hashes-per-second.
- α Attacker's hash-ratio, as a fraction: $\frac{A}{H+A}$.
- b A single block.
- R The reward received for mining a block, in tokens.
- d Mining difficulty.
- H Total honest hash-rate, in hashes-per-second.
- t The difference between the timestamp of some block and its direct main-chain ancestor, in seconds.
- F The transaction fees earned when mining a block, in tokens.
- u A bit which is equal to True iff the current tip of the main-chain has at least one uncle.

ACRONYMS

ASIC application specific integrated circuit
block-DAG block directed-acyclic-graph
CPU central processing unit
DAA difficulty-adjustment algorithm
DeFi decentralized finance
EIP Ethereum improvement proposal
geth Go Ethereum
HUJI Hebrew University of Jerusalem, Israel
MDP Markov decision process
MEV miner-extractable value
ML machine learning
PoS proof-of-stake
PoW proof-of-work
PUM preemptive uncle maker

RAM random-access memory
RPC remote procedure call
RUM riskless uncle maker
SSD solid state drive
TD total difficulty
URL uniform resource locator