The Attack of the Clones Against Proof-of-Authority

Parinya Ekparinya
University of Sydney

Vincent Gramoli
University of Sydney and Data61-CSIRO

Guillaume Jourjon
Data61-CSIRO

Abstract

The vulnerability of traditional blockchains have been demonstrated at multiple occasions. Various companies are now moving towards Proof-of-Authority (PoA) blockchains with more conventional Byzantine fault tolerance, where a known set of n permissioned sealers among which no more than t are Byzantine seal blocks that include user transactions. Despite their wide adoption, these protocols were not proved correct.

In this paper, we present the Cloning Attack against the two mostly deployed PoA implementations of Ethereum, namely Aura and Clique. The Cloning Attack consists in one sealer cloning its key-value pair into two distinct Ethereum instances that communicate with distinct groups of sealers. To identify their vulnerabilities, we first specified the corresponding algorithms. We then infer the topology of the largest PoA network, POA Core, through active measurement. We deploy one testnet for each protocol and demonstrate the success of the attack. Finally, we propose a simple counter-measure that prevents an adversary from double spending.

1 Introduction

Ethereum is one of the most popular blockchain systems thanks to the large ecosystem of distributed applications that it executes. Unfortunately, the default Ethereum protocol based on *proof-of-work (PoW)* forks because it allows distinct blocks to be appended at the same index of the chain. This fork situation can lead to security vulnerabilities, like double spending, if it is not detected early enough [9, 16, 17]. Alternative protocols, called *proof–of-authority (PoA)* protocols, that aim at avoiding forks have recently been integrated in the mostly deployed versions of Ethereum, parity and geth, and are currently used world-wide. Yet, to our knowledge, the level of security offered by PoA protocols has not been assessed.

These PoA consensus protocols, called Aura and Clique, are said to use a proof-of-authority because they restrict the creation of a block to a fixed set of n authority nodes, called *sealers*, among which a maximum of $t < \frac{n}{2}$ can misbehave or be *Byzantine*. They aim at solving the well-known Byzantine consensus problem [12], where among n nodes the correctly behaving ones agree on a unique block despite the presence of $t < \frac{n}{2}$ Byzantine nodes. PoA gives the *sealers* the authority to *seal* a block, which consists of signing cryptographically the block. This set of sealers can possibly change over time if a subset of the participants allow it, hence being well suited for dynamic consortia of participants. This PoA is an appealing alternative to PoW for industry sectors with an interest in limiting resource wastes and in increasing security.

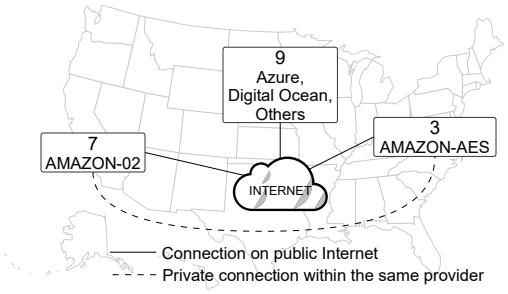

Figure 1: Topology of sealers in the POA Core network

For these reasons, PoA recently gained rapid momentum in critical applications. Industrials, like Lavaa, propose a tracking service to prevent fraud counterfeiting on top of Aura [20]. Microsoft describes how to deploy Aura "in production".[1] Amazon Web Services offers PoA through the Clique protocol built in geth to its customers.[2] They implemented a service that aims at maintaining data privacy and integrity in a multi-tenant scenario. Every day, Internet users exchange digital assets through multiple instances of these two protocols. Rinkeby is a network of 65 participants offering the Clique service across four continents to its users.[3] Sokol and Kovan

[1] https://docs.microsoft.com/en-us/azure/blockchain/templates/ethereum-poa-deployment.

[2] https://aws.amazon.com/blogs/apn/launch-enterprise-ready-blockchain-networks-\on-aws-in-minutes-with-kaleido-a-consensys-solution/.

[3] https://www.rinkeby.io/#stats.

are other Ethereum testnets running the Aura protocol.[4] In particular, Sokol is used to test features before launching them on the largest in-production network called POA Core. Figure 1 illustrates the sealers topology of POA Core that we inferred through active measurement and sealer fingerprinting (cf. Section 8 for details). An interesting theoretical work by De Angelis et al. [1] indicated that the consistency of Aura can be limited, for example if the clocks are far apart, and that Clique is eventually consistent. To the best of our knowledge, these algorithms have not been formalized and it is unclear whether an attacker could violate data integrity.

In this paper, we show that, under specific conditions, PoA is not secure. To this end, we design, implement and experiment an attack, called the Cloning Attack, against both Ethereum/Aura and the Ethereum/Clique protocols that allow to steal digital assets and propose ways to alleviate the vulnerability. The Cloning Attack consists of a sealer attacker cloning a public-private key pair to convince half of the honest sealers that a transaction is correctly committed before erasing this transaction to double spend its coins. Thanks to the cloning, to convince half of the honest sealers that transactions are committed, the attacker simply needs to delay messages between two halves of honest sealers. Note that this is achieved without knowing the precise locations of sealers but simply through OS and network fingerprinting that allows to identify the autonomous systems the sealers belong to. In the POA Core example depicted in Figure 1, it is sufficient for an Amazon sealer to clone itself on Azure and leverage a 30 second message delay between AWS and the rest of the network to commit conflicting transactions and double spend.

We demonstrate the effectiveness of the Cloning attack by double spending in two testnets, one running `parity` and the other running `geth`. The application of the Cloning attack to Aura is slower as it consists of the attacker sealing more blocks in one branch while its application to Clique is faster but more subtle as it consists of the attacker disordering the victim sealers to minimize the weight of a branch. Overall, we found that Aura required lesser topological knowledge for a malicious sealer to achieve double spending with 100% success rate when compared to Clique. The attack against Clique is about twice faster but its success rate ranges from 60% to 100%. In order to remedy the identified vulnerabilities, we propose to modify these two PoA protocols to preserve their safety. Even though our counter-measures introduce liveness limitations in these algorithms they make them more suitable for critical applications.

Section 2 presents the related work. Section 3 describes the model. Section 4 formalizes both Aura and Clique protocols as implemented in respectively `parity` and `geth`. Section 5 describes the Cloning Attack against both consensus algorithms. Sections 6 and 7 explain how to exploit it to double spend in Aura and Clique, respectively. Section 8 indicates how we inferred the necessary topological information of the currently running POA Core network. We then present in Section 9 our evaluation of the Cloning Attack on both protocols, while Section 10 discusses our results and potential counter-measures. Section 11 concludes the paper.

[4] https://kovan-testnet.github.io/website/.

2 Background

Most of the known *double spending* attacks against blockchains exploit their inherent permissionless mechanism by including in the blockchain a transaction that transfers coins and then discard this transaction, hence allowing to re-spend the previously spent coins in a subsequent transaction. Below we list some of these attacks to explain the recent raise of alternative protocols based on PoA.

Perhaps the most conventional way to double spend in permissionless blockchains is for an attacker to exploit more than half of the mining power of the system to create an heavier or longer branch that can overwrite transactions that were expected to be sufficiently confirmed or committed [18]. In some blockchains, a quarter of the mining power appears enough in theory to attract participants into a coalition whose cumulative mining power reaches strictly more than half of the total mining power [7]. SmartPool [13] copes with the centralisation of mining power into these blockchains and the risk of mining pools to join a coalition of strictly more than half of the total mining power.

To attack permissionless blockchains without a significant mining power, researchers attacked the network. The Eclipse attack against Bitcoin [9] consists of isolating at the IP layer a victim miner from the rest of the network to exploit its resources. The Blockchain Anomaly [15] exploits message reordering in Ethereum to abort transactions that seemed sufficiently confirmed. The Balance Attack [16] partitions the network into groups of similar mining power to influence the selection of the canonical chain. Recently, actual man-in-the-middle attacks were run to demonstrate the feasibility of stealing assets in Ethereum without a significant mining power [6].

To cope with these attacks, alternative blockchains have offered to reach Byzantine consensus using proof-of-stake sometimes probabilistically [8, 14], sometimes deterministically [11, 19]. Given the large literature on Byzantine fault tolerance [12], we know that leveraging authentication, upper-bounding the number of failures with $t < \frac{n}{2}$ and assuming *synchrony* or delivering messages in a known bounded time should allow participants to reach a consensus on a unique block at a given index of the chain, hence avoiding double spending. What is key for critical applications is that these Byzantine fault tolerant blockchains guarantee that no participants double spend even when messages get unexpectedly delayed.

Proof-of-Authority (PoA) was recently proposed as a Byzantine fault tolerant consensus mechanism that integrates with

the Ethereum protocol [10]. It is available in the two major Ethereum implementations, called geth and parity, under the name of the Clique and Aura protocols, respectively. The concept is similar to traditional Byzantine fault tolerant consensus in that only n sealers are permissioned to create new blocks but requires authentication and strictly less than $\frac{n}{2}$ Byzantine participants, similarly to the seminal work on Byzantine consensus [12]. Although it does not necessarily improve the protocol performance, its lack of double spending raised interest from the industry [20]. Some recent study [1] of PoA revealed that unsynchronized clocks could affect Aura's consistency whereas Clique was only eventually consistent, however, to our knowledge no attacks have been proposed against Aura or Clique until now.

3 Model

We consider a distributed system of n permissioned sealers whose identifiers are $p_1, .., p_n \in Ids$. As the blockchain is open, it accepts the requests issued by nodes that are not necessarily sealers, hence the overall number of participants can be larger than n, but only n participants can propose blocks and seal (or sign) them. We assume authentication through a public-key cryptosystem that allows participants to easily identify that a block is correctly signed by a sealer so that incorrectly signed blocks are simply ignored. We assume that keys cannot be forged or stolen by Byzantine participants and that appropriate private keys are correctly distributed to the sealers initially. As in the Dolev-Yao model [4], we assume the attacker, who has the control over the Byzantine participants, can intercept messages.

It is claimed that the Aura algorithm is designed to "tolerate up to 50% of malicious nodes" [10], however, in general a participant cannot decide if half of the nodes pretend that a block is decided while the other half of the nodes pretend that the same block is not decided [12]. This is the reason why we assume in this paper that no more than $t < \frac{n}{2}$ participants can be *Byzantine* and can act in an arbitrary way, hence a majority of participants always remain *correct*.

As it is well known that consensus cannot be reached in an environment where communication is asynchronous in the presence of faults, it appears natural to assume additional synchrony. It is unclear whether PoA protocols can be safe under *partial synchrony*, where message gets delivered in a bounded amount of time that is not known from the algorithm [5] or how long communication can take for these protocols to work. As an example, a preliminary version of Aura was mentioned to require synchrony in a web document[5], however, this information appears outdated as the implementation is closer to another documentation [10] that does not mention this, as we explain in Section 4.1.

The questions we investigate is whether PoA protocols work under partial synchrony, and if not, whether the risk of unexpected message delays is benign (termination of the consensus is not guaranteed but no double spending occur) or can have dramatic consequences (disagreement can occur), letting an attacker double spend. Interestingly, we actually demonstrate in PoA protocols that the worse can happen in case of unexpected message delays.

[5] https://wiki.parity.io/Aura.

4 PoA Consensus Algorithms

In this section we describe the two main variants of PoA algorithms, called Aura and Clique, popularized by the predominant Ethereum software, called parity and geth, respectively. We first formalize two distinct versions of the Aura algorithm that are both publicly available online. We then formalize the Clique algorithm.

4.1 The Aura consensus algorithms

There exist two distinct versions of the Aura algorithm as documented online, one that corresponds to the current parity implementation of the Ethereum protocol and another [10] that uses rounds to decide whether a consensus decision is reached.

4.1.1 The parity Aura algorithm

Algorithm 1 depicts the way Ethereum/Aura guarantees that participating nodes reach consensus on the uniqueness of the block at a given index of the blockchain as implemented in Parity-Ethereum-v2.0.8 (v2.0.8 was the latest version at the time we performed our experiments). Every participating node maintains a state comprising a set of *sealers*, its current view of the blockchain c_i as a directed acyclic graph $\langle B_i, P_i \rangle$ and a block b with fields *parent* that links to the parent block, a *sealer* and a *step* indicating the time at which the block is added to the blockchain, as explained below. Initially, they are $\perp$ meaning "undefined".

The function propose() is invoked in order to propose a block for a particular index of the blockchain. The consensus is reached once the block is decided which can happen much later as we will explain in the function is-decided() (Line 27) below. The algorithm discretises time into steps that corresponds to consecutive periods of step-duration time, as specified in a configuration file. Each sealer executes an infinite loop that periodically checks whether the clock-time() indicates that this is its turn to propose a block (Line 13). When it is its turn (Line 14), a sealer sets the parent of the block to the last block of its view and signs it (Line 36).

Each broadcast() invoked by the propose() function sends messages that get delivered to all other participating nodes that are correct. The deliver() function (Line 23) is thus invoked at each participating node, regardless of whether it is a sealer, upon reception of the broadcast message. Once a blockchain

Algorithm 1 The parity Aura algorithm at process p_i

```
1:  State:
2:    sealers ⊆ Ids, the set of sealers
3:    c_i = ⟨B_i, P_i⟩, the local blockchain at node p_i is a directed
4:      acyclic graph of blocks B_i and pointers P_i
5:    b, a block record with fields:
6:      parent, the block preceding b in the chain, initially ⊥
7:      sealer, the sealer that signed block b, initially ⊥
8:      step, the step of the blockchain when the block gets added, initially
      ⊥
9:    step-duration, the duration of each step as configured
10:
11: propose()_i:
12:   while true do
13:     step ← clock-time()/step-duration                  ▷ discretize time
14:     if i ∈ sealers ∧ step mod |sealers| = i) then      ▷ my turn
15:       b.parent ← last-block(c_i)                       ▷ link a block
16:       b.sealer ← p_i                                   ▷ seal the block
17:       broadcast(⟨{b}, {b.parent}⟩)                     ▷ send the block
18:     sleep(step-duration)                               ▷ wait before looping
19:
20: score(⟨B_i, P_i⟩):
21:   return UINT128_MAX × height(⟨B_i, P_i⟩) − step-num(⟨B_i, P_i⟩)
22:
23: deliver(⟨B_j, P_j⟩)_i:
24:   if score(⟨B_j, P_j⟩) > score(⟨B_i, P_i⟩) then
25:     ⟨B_i, P_i⟩ ← ⟨B_j, P_j⟩
26:
27: is-decided(b)_i:
28:   V ← {b_k.sealer | b_k ∈ B_i; k ≥ i}                  ▷ sealers in blocks since b_i
29:   return (|V| × 2 > |sealers|)                         ▷ more than majority of sealers signed
```

view is delivered to participant i, the node compares the score of the blockchain view it maintains to the blockchain view it receives using the score (Line 20). The highest blockchains has the greatest score, however, if two blockchains share the same height, then the one that is denser in terms of its number of non-empty slots obtains the highest score. This is indicated by the two functions height and step-num that represents the height of the blockchain and the number of slots for which there exists a block in the blockchain.

4.1.2 Round-based variant of the Aura algorithm

The Aura algorithm implemented in parity is not the only algorithm called, Aura. Another variant is presented in the PoA Network white paper available online [10]. Algorithm 2 presents the different decision process of this variant, the rest of the pseudocode being identical to Algorithm 1.

In order to know whether a block b is *decided* at the end of a successful consensus (Algo 2, Line 1), a participant simply has to check whether there exist two consecutive rounds following block b, in each of which the blocks are sealed by a majority of the sealers.

Note that while presented in some documentation, this alternative Aura specification is not the one used by the mainstream implementation of the protocol. The current definition of the Aura algorithm disregards the rounds and simply requires enough blocks to be sealed[5]. Although the version of Aura we experiment in this paper is the mainstream one (Algorithm 1), the attack we present in Section 6 could also be applied to this more restrictive definition presented in Algorithm 2.

Algorithm 2 The round-based variant of Aura at process p_i

```
1: is-decided(b)_i:
2:   ℓ ← |sealers|                                  ▷ number of validators
3:   round1 ← (b.step, b.step + ℓ]                  ▷ steps in next round
4:   round2 ← (b.step + ℓ, b.step + 2 * ℓ]          ▷ steps in the 2nd next round
5:   maj1 ← |{b' : b'.step ∈ round1}| > ℓ/2         ▷ majority in round 1
6:   maj2 ← |{b'' : b'.step ∈ round2}| > ℓ/2        ▷ majority in round 1
7:   return (maj1 ∧ maj2)                           ▷ decided if majority in both rounds
```

4.2 The Clique consensus algorithm

Algorithm 3 depicts the pseudocode of the Ethereum/Clique consensus algorithm. This solution is the one used currently in geth-1.8.20-stable.

Every participating node shares the same initial block, the genesis block, which also contains the block-period, the period between consecutive block creations. Similarly to the Aura protocol, each node maintains its own view of the growing blockchain c_i as a directed acyclic graph $\langle B_i, P_i \rangle$. A block b contains a *number* as an index of the block in the blockchain, a *weight* as a weight of the block, a *parent* field that links to its parent block and a *sealer*.

The propose() function runs an infinite loop in order to propose blocks to the blockchain when certain conditions are satisfied. The first condition (Line 26) requires the process to wait for blocks from other sealers until none of the last sealer-limit blocks contain its signature. In the current implementation the sealer-limit must be $\lfloor |sealers|/2 \rfloor + 1$, which is the smallest majority. As a result of this first condition, sealers need to take turn to sign blocks. The second condition (Line 28) is to wait for block-period.[6] When both conditions are met, the process checks if it is its turn to sign the block (Line 29). The process may sign a block right away with *weight* equal to 2; otherwise, it may sign a block with *weight* equal to 1 after a random delay between 0 and $500 \times \lfloor |sealers|/2 \rfloor + 1$ milliseconds (Line 32). The consensus is reached once the block is decided later as we will describe in the function is-decided() (Line 46). The last step in the loop, broadcast(), sends messages to other processes.

Upon reception of the broadcast message, the deliver() function (Line 42) is invoked at each participating node regardless of whether it is a sealer. The total-weight function (Line 39) used by the process compares the weight between two blockchain views, a current blockchain that it maintains locally and the one freshly received. The process updates its

[6]The default block-period is 15 seconds as developers suggest the same duration to remain analogous to the proof-of-work blockchain Ethereum.

Algorithm 3 The Clique PoA blockchain consensus algorithm at process p_i

```
1:  State:
2:    sealers ⊆ Ids, the set of sealers
3:    c_i = ⟨B_i, P_i⟩, the local blockchain at node p_i is a directed
4:      acyclic graph of blocks B_i and pointers P_i
5:    b, a block record with fields:
6:      parent, the block preceding b in the chain, initially ⊥
7:      sealer, the sealer that signed block b, initially ⊥
8:      number, the index of the block in the chain, initially ⊥
9:      weight, the weight of a block, initially ⊥
10:   block-period, minimum duration in second between timestamps of
11:     two consecutive blocks, initially 5 seconds
12:   majority, the number of ⌊|sealers|/2⌋ + 1
13:   sealer-limit, maximum number of consecutive blocks among which a sealer
14:     can sign at most one block, initially set to the majority
15:
16: sign-recently(c_i, n)_i:
17:   λ ← sealer-limit
18:   ret = false
19:   for m = n − λ, ..., n do                          ▷ loop through last λ blocks
20:     if b_m.number mod |sealers| = i then ret = true
21:   return ret
22:
23: propose()_i:
24:   while true do
25:     n ← last-block(c_i).number                      ▷ last block index
26:     wait until ¬sign-recently(c_i, n)               ▷ wait until I can seal a block
27:     T ← get-last-timestamp(c_i)
28:     wait until clock ≥ T + block-period             ▷ wait ≥ block-period
29:     if (n + 1) mod |sealers| = i then               ▷ in-order sealing
30:       b.weight = 2                                  ▷ block weight 2
31:     else                                            ▷ out-of-order sealing
32:       sleep(rand([0, 500 × majority]) ms)           ▷ random delay in milliseconds
33:       b.weight = 1                                  ▷ block weight 1
34:     b.number = n + 1                                ▷ increment block index
35:     b.parent ← last-block(c_i)                      ▷ link a block
36:     b.sealer ← sign()                               ▷ seal the block
37:     broadcast(⟨{b}, {b.parent}⟩)                    ▷ send the block
38:
39: total-weight(⟨B_i, P_i⟩)_i:                         ▷ total weight
40:   return ∑ b.weight | b ∈ B_i
41:
42: deliver(⟨B_j, P_j⟩)_i:
43:   if total-weight(⟨B_j, P_j⟩) > total-weight(⟨B_i, P_i⟩) then   ▷ heaviest
44:     ⟨B_i, P_i⟩ ← ⟨B_j, P_j⟩
45:
46: is-decided(b)_i:
47:   V ← {b_k.sealer : b_k ∈ B; k ≥ i}                 ▷ sealers in blocks since b_i
48:   return |V| > majority                            ▷ more than majority of sealers signed
```

local view if the received blockchain is heavier; otherwise it keeps the same local blockchain view.

To consider whether a block b is *decided* (Line 46), a process has to check the set of sealers who sign blocks after b. Only when a majority of sealers have appended subsequent blocks to the chain, can a block be considered decided.

5 The Cloning Attack

In this section, we present the cloning attack to double spend in PoA blockchains. In particular, we present the commonalities between the attacks against Aura and Clique, namely the cloning process that allows an attacker to play different roles in the blockchain, the majority that allows two groups of sealers to make progress without the other, and the way transactions should conflict to double spend. The difference in how these attacks are applied to Aura and Clique are deferred to Sections 6 and 7, respectively.

By assumption, only a minority of the sealers can be malicious, this is the reason why PoA algorithms require a majority of sealed blocks to consider whether a block and its transactions appear to be committed. Intuitively, this should prevent the malicious sealers to form a coalition that can double spend. In reality, as we explain below, $(2 - n \bmod 2)$ attacker(s) cloning their own instance into two clones are sufficient to double spend.

5.1 Cloning instances by duplicating keys

The first step necessary in in the Cloning Attack is for some attacker to duplicate its Ethereum instance into two clones. *Cloning* consists for a single user, the attacker, of running two instances of the Ethereum protocol with the same *address* or public-private key pair. Note that these two instances could run either on the same machine, using the same IP address, or on distinct machines with distinct IP addresses. We call these two instances *clones* because one has the same information as the other before messages start being delayed. In addition, during the whole duration of the attack, both clones use the same public-private key pair. Interestingly, we noted that Ethereum allows these two cloned instances to mine blocks, however, as they use the same private key to seal blocks, they are considered to act as a unique sealer.

At some point, the attacker exploits message delays (either accidental or as a result of a network attack [6]) between two groups of a minority of $\lceil n/2 \rceil - 1$ sealers, hence creating a transient partition. At this moment, the two clones may not share exactly the same database content as they may not be aware of the exact same blocks that are present in the blockchain. To maintain the cloning at the start of the partition, the attacker copies the content of the blockchain database of one of the clone to the database of the other clone and connects each of these clones to a different partition. During the time of this paritition, the Ethereum protocol readjusts the peering so that sealers within the same group keep communicating.

Note that they are various ways of obtaining a partition in the Ethereum network either by misconfiguring routes, leveraging natural disasters or maliciously attacking the network. One example of a network attack that allows to partition the Internet is the BGP hijacking attack. It works by having an attacker advertising to one group wrong routes that reach the

other group in order to intercept all traffic between the two groups. Once the traffic is rerouted, the attacker can simply delay the propagation of messages. We refer the interested reader to previous work [6] that focuses on man-in-the-middle attacks against Ethereum permissioned networks.

5.2 Majority groups to guarantee progress

Clones are exploited in the attack to give the illusion to correct sealers that each group contains a majority of sealers. In order to progress towards a double spending situation, each group must commit transactions and thus decide blocks, this is why we need $(2-n \bmod 2)$ attackers that clone instances.

- **Case n is odd.** The correct sealers can be split into two groups of $(n-1)/2$ sealers, each representing a minority. In order to guarantee progress of the protocol on both sides of the partition, a single attacker can simply add one clone in each minority, hence reaching a majority of $\lfloor n/2 \rfloor + 1$ sealers on each side. This is the reason why $(2-n \bmod 2) = 1$ attacker is sufficient when n is odd.
- **Case n is even.** The attacker will end up splitting the $n-1$ correct sealers into two groups of different sizes, one that contains $n/2$ sealers and another that contains $n/2-1$ sealers. While it is sufficient to include a clone in the former group to guarantee their progress, one must add at least two clones to the latter group to guarantee their progress. This is the reason why $(2-n \bmod 2) = 2$ attackers are necessary when n is even.

To conclude, the $(2-n \bmod 2)$ attacker(s) thus partition a network of n sealers into roughly two halves to which they add clones so that each group contains a majority of at least $\lfloor n/2 \rfloor + 1$ sealers. This guarantees the progress of the protocol on each group so as to obtain the commit of a transaction TX_1 on one group and the canonical chain containing TX_2 in the other group. For example, there must be at least 5 sealers in each subgroup for a network of $n = 9$ sealers. Such a condition is required to ensure termination of the consensus algorithm, so that blocks will be decided, or appear to be final, from the viewpoint of both subgroups.

Note that we consider here the necessary time for a partition. In a realistic scenario, the attacker may want the effect of its transaction to take effect before stopping the partition. For example, an attacker buying a good in transaction TX_1 may want to receive the good before the transaction gets discarded from the blockchain.

5.3 Conflicting transactions

The most common way of executing a double spending is to make sure a transaction TX_1 ends up being included in one branch of a fork, then convincing the recipient that TX_1 is committed, before resolving the fork by discarding the branch of this transaction TX_1. Later on, the sender of the transaction TX_1 can simply reuse the coins he initially spent in TX_1 in another transaction TX_2. Interestingly in Ethereum, if the conflicting transaction TX_2 is not issued early enough, then TX_1 could be re-included in a mempool and re-committed later on.

The goal is for the clones to leverage the partition to rapidly issue two conflicting transactions. As soon as the blockchain network is divided into two subgroups, the attacker issues a minimum of two conflicting transactions, at least one transaction to each subgroup. A typical example to illustrate the double spending attack is two conflicting transactions:

TX_1 where Alice gives all her coins to Bob in the first transaction sent to one group and

TX_2 where Alice gives all her coins to Carol on the other transaction sent to the other group.

It is clear that committing both transactions would violate the integrity of Alice's account and would result in a double spending. Once the first transaction appears committed, ending the partition will have the effect of discarding one of the two transactions.

In the next two sections, we explain how the majority of sealers in Aura and the order of the sealings in Clique allow to select the transaction to be discarded by the system.

6 The Cloning Attack Against Aura

We now present a simple way to apply the Cloning attack to double spend in Aura. To discard the branch, say the *victim branch*, that contains TX_1 and double spend, the attacker must influence Aura to select the branch containing TX_2, say the *attacker branch*, as the canonical chain.

As explained earlier in Algorithm 1, the current implementation of Aura simply chooses the longest chain as the canonical chain whenever a fork is detected. So, to influence the selection of the attacker branch as the canonical chain, the attacker simply has to contribute to the attacker branch by sealing more blocks in the group maintaining this branch than the other group.

To seal more blocks in one branch than another, the attacker maintains the partition during $(n+1) \times s$ seconds, where n is the number of sealers and s is the step duration in seconds that separates consecutive blocks. The reason is twofold.

- First, as mentioned earlier in Algorithm 1, Aura requires ns delay after a block is created to ensure that it is decided. Deciding a block on the victim side is necessary to make sure that TX_1 gets committed. Given that both the size of the group on each side is $\lfloor n/2 \rfloor + 1$ and that each sealer seals one after another, the attacker clone must also seal at least one block.

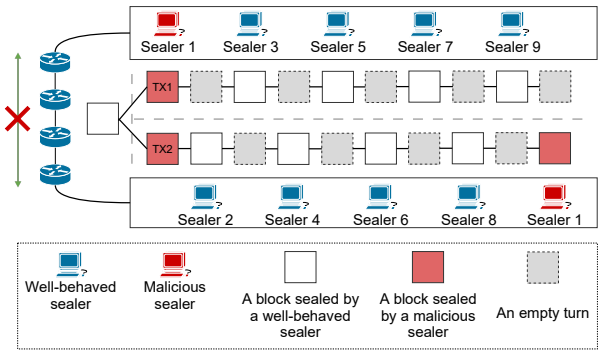

Figure 2: Applying the cloning attack to double spend in Aura requires the attacker, "Sealer 1", to delay messages during $(n+1) \times s$ seconds for transaction TX_1 to be committed on the upper branch and for the attacker to seal more blocks on the lower branch than on the upper branch

- Second, the attacker must ensure that the attacker branch is longer than the victim branch so that the attacker branch gets selected by Aura as the canonical branch. This can only be done if the attacker seals two blocks on the attacker branch, i.e., one extra block compared to the number of blocks it sealed on the victim side. As a result, the attacker needs to maintain the network partition for $(n+1) \times s$ seconds to get at least two turns in which it can seal a block.

Example with 9 sealers. For the sake of simplicity, Figure 2 depicts the cloning attack against Aura with a network partition where there are $n = 9$ sealers and where $2-n \bmod 2 = 1$ sealer is malicious, namely "Sealer 1". This attacker is thus present in both groups through its two cloned instances and gives the illusion that each group contains a majority of $\lfloor n/2 \rfloor + 1 = 5$ sealers while one of the sealers in each group is actually a clone. As we can see, this attack translates into having Sealer 1 creating the last block (depicted in red in the figure) only on the lower partition before merging the two partitions. By doing so, Sealer 1 makes sure that this branch will be the canonical branch whereas the upper branch will disappear. The attackers is thus guaranteed to double spend successfully.

7 The Cloning Attack Against Clique

In this section, we apply the cloning attack against Clique. In Clique, the Cloning Attack does not require to take as long as in Aura. Unlike in Aura, a sealer of Clique can seal a block even when it is not its turn. Depending on their turn, some sealers may have to wait while others do not. These differences impact the way the attacker can influence the selection of one branch of a fork as the canonical chain and allow an attacker to double spend faster than in Aura.

7.1 In-order and out-of-order sealers

The cloning attack against Clique differs from the one against Aura in the moment at which it starts delaying messages. Because the order of sealing is important in Clique, the attacker should ideally decide to start delaying the messages based on the sealer's turn to seal a block.

When a sealer seals a block while it is his turn, we call this sealer an *in-order sealer* and the block an *in-order block* (cf. Alg. 3, Line 29). There is at most one in-order sealer to seal the current block in each partition of Clique. When a sealer seals a block while it is not his turn, we call this sealer an *out-of-order sealer* and this block an *out-of-order block* (cf. Alg. 3, Line 32). As a sealer must wait for *sealer-limit* blocks between two blocks it seals, there are at most $(n - \textit{sealer-limit})$ potential out-of-order sealers to seal a block. The in-order block contributes a weight of 2 to the weight of its branch whereas the out-of-order blocks contribute to 1 to the weight of its branch, hence sealing in-order or out-of-order impacts the decision regarding the branch selection process.

In addition, an in-order sealer can append a block to the chain without waiting for any delay as shown in Line 29 of Alg. 3. By contrast, an out-of-order sealer has to wait for a random period as indicated at Line 32 of Alg. 3. This mechanism gives the in-order sealer some time to be the first to seal a block in his turn, but allows out-of-order sealers to seal a block if the in-order sealer is lagging.

As the canonical chain is chosen among the branches of a fork by comparing the sum of their block weights, the attacker must have a maximum number of in-order sealers at the time of the partition to maximize the overall weight. Hence, to influence the selection of the branch as the canonical chain, the attacker must choose the proper turn to start delaying messages. If not done properly, the attackers risks to maximizing the weight of the branch where its transaction was included, limiting the chances of a successful double spending.

7.2 Disordering sealers to select a branch

Figure 3 depicts the execution of the attack with $n = 9$ sealers and one attacker (Sealer 1) as time increases from top to bottom. Initially, the blockchain starts with block 5, indicating that the first block is sealed by Sealer 5. As times goes on, Sealers 6, 7, 8 and 9, seal one after the other the subsequent blocks of the blockchain. As there is no partition yet, the in-order sealers are the first to sign these blocks during their respective turn, hence all blocks are in-order blocks represented in blue in the figure. Next to each created block is a list of sealers that are either unable to seal (grey), in-order sealers (green) or out-of-order sealers (yellow).

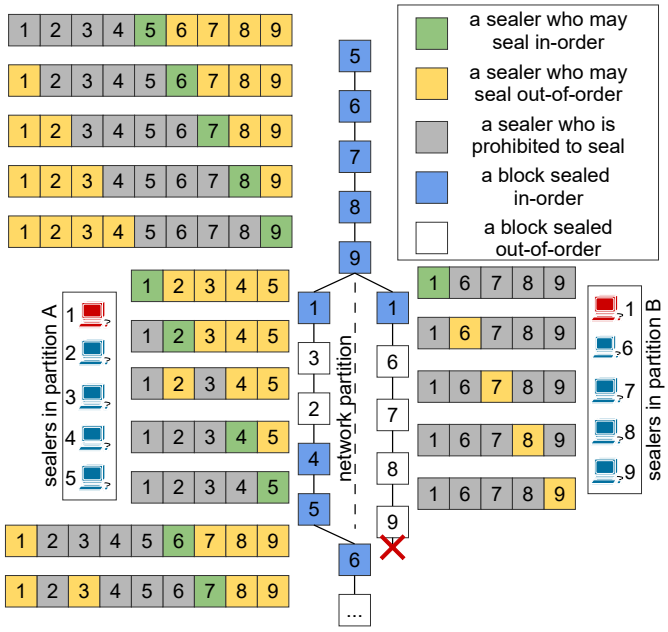

Figure 3: An execution of the in-order cloning attack against Clique where $n = 9$ sealers mine the blue blocks in-turn before the messages get delayed, after which each group seals five blocks, 3 in-order blocks on the left group and 1 in-order block on the right group

Consider that Sealer 1, the attacker, performs the cloning and delays the network messages. Right after Sealer 9 sealed his block, Sealer 1 starts intercepting the messages between the group of sealers 2, 3, 4 and 5 on the left side and the group of sealers 6, 7, 8 and 9 on the right side. Note that Sealer 1 is represented on both sides because of the presence of one of its clones on each side. The resulting partition is indicated in Figure 3 with a fork of the blockchain into two branches. Right after the partition starts, Sealer 1 issues two conflicting transactions TX_1 and TX_2 on each side of the partition that will double spend. The two clones of Sealer 1 allow him to seal one block in each group. Note that these blocks are labelled 1 and represented in blue because Sealer 1 is the in-order sealer at this point in time. After sealing, Sealer 1 is no longer able to seal any block due to the *sealer-limit*, hence Sealer 1 is depicted in grey in both groups.

On the right side of the partition, we can see that Sealer 6 seals the following block, even though it is not the in-order sealer at this moment. This is because the in-order sealer, Sealer 2, cannot communicate with this group as the network is partitioned. For some reason it might also be the case on the left side of the partition that Sealer 2 is not fast enough to seal the next block and that another sealer, say Sealer 3, manages to seal it before. Note that this can happen as the delay Sealer 3 has to wait before sealing is a random number that can be null (cf. Alg. 3, Line 32). However, this last seal from Sealer 3 prevents it from sealing the next block in-order as it has to wait for the *sealer-limit*, hence the next block is again out-of-order. The process continues where sealers on the left side seal in-order whereas sealers on the right side seal out-of-order.

Finally, the attacker does no longer need to delay the messages and can stop the partition as both transactions TX_1 and TX_2 are now successfully committed. In fact, the transactions are both now in the first block of a series of $\lfloor n/2 \rfloor + 1 = 5$ consecutive blocks, which is sufficient for all Clique users to consider these transactions as committed because their block is decided as indicated at Line 48 of Alorighm 3. We can conclude that the weight gained by the branch on the left side during the partition is $3 \times 2 + 2 \times 1 = 8$ because it contains 3 in-order blocks and 2 out-of-order blocks. By contrast, the weight gained by the branch on the right side during the partition is $1 \times 2 + 4 \times 1 = 6$ because it contains 1 in-order block and 4 out-of-order blocks. It follows from the difference in weight of the two branches that the heaviest branch on the left side is chosen as the canonical branch whereas the lightest branch on the right side is simply discarded by the protocol (cf. Alg. 3, Line 43).

7.3 Attack regardless of the order of sealers

Note that even if the attacker does not know the topology, there is a way to attack Clique. The attack is slightly different from the previous one as it relies on the possibility for the attacker to become the only sealer able to seal a block on both sides of the partition. The attacker can simply seals a single block on the victim branch, and keep sealing blocks on the attacker branch. In the worst case scenario for the attacker, all the $\lfloor n/2 \rfloor + 1$ upcoming in-order sealers end up on the victim side, which will maximize the weight of the branch on the victim side gained during the partition. Recall that the *sealer-limit* is always $\lfloor n/2 \rfloor + 1$ in Clique (Alg. 3, Line 14), Now, if the attacker stops sealing a second block on the victim side, then the maximum weight gained on this side during the partition will be $(\textit{sealer-limit} \times 2)$. The attacker simply needs to keep sealing on the other branch until the gained weight on this branch reaches $(\textit{sealer-limit} \times 2 + 1)$. In this case, the attacker successfully double spends regardless of the sealer turn in each group.

8 Inferring the POA Core Topology

In this section, we show how to infer the topological information necessary to run the Cloning attack against the currently running POA Core network. Since POA Core is a public blockchain, the network requests its sealers to comply with some eligibility requirements as part of an application procedure. An important restriction is that all individuals running a sealer node must be US residents; these individuals are also

requested to deploy their Ethereum sealer instance within the US. To bootstrap the POA Core network, the first sealer of a network called the "Master of Ceremony" generates the initial keys and distributes them to a group of independent participants; together they form the first group of sealers that govern the blockchain network. This governance may vote for adding new sealers or removing existing ones to limit, for example, bribery attacks. Before a new sealer candidate can be eligible for a sealer role on POA Core network, one needs to apply a sealer role on Sokol, a testnet for POA Core network, and actively participate in its on-chain governance for some minimum period of time. The conditions also require a sealer candidate to be a public notary in the US with a valid license; this is not a hard requirement for Sokol, but it is mandatory for POA Core network. Furthermore, sealer individuals need to write posts about themselves in a POA forum; these posts often include their full name and their notary public licenses, which can then be used to retrieve their mail addresses.

Netstat information. In addition to the public information part of the sealer application, POA Core employs two monitoring tools that also publicizes information about its sealers. For the public to be able to browse blockchain information conveniently, POA Core has its own block explorer site[7], which reveals the sealing order of all the active sealers as well as the required time to seal one block. In order to monitor the system and the network status of each sealer, POA Core incorporates the netstat server within its own web-based dashboard site[8] and displays the information received from all participating sealers. Some of this information, including the number of peers of each sealer, its OS names and its latency allowed us to uniquely identify the sealer nodes.

Active measurement and fingerprinting. In order to obtain the versions of the parity protocol run by sealers, their IP addresses, and their port numbers, we instrumented the code of Ethereum parity and launched a new node participating in POA Core. Using the information gathered from the public sources, we inferred the topology of the POA Core network to delay message for the Cloning attack. We used the OS names obtained from the netstat server dashboard in order to guess the datacenters where the nodes are deployed. Assuming the sealers reported correct information, the suffix of the OS names reported to the netstat server indicates the cloud provider where the node runs, for example aws indicates Amazon Web Service while azure indicates Microsoft Azure. The information gathered by our node from its peers included non-sealer nodes such as boot nodes, so we eliminated some of these non-sealer nodes by examining their port numbers and software versions. For instance, it is trivial that a node that runs Parity on tcp/21000 is not a sealer, as the netstat dashboard indicates that all the sealers use tcp/30303. Next, we simply extracted the autonomous system numbers (used to route the Internet traffic globally) based on their list of IP addresses. After that, we used nmap[9] to perform network OS fingerprinting on a list of IP addresses that we had already obtained. Although the results from network OS fingerprinting is not 100% accurate, it was sufficient to map this information back to the OS names obtained from the dashboard because the cloud providers often use different kernel versions in their OS images. Finally, we estimated the datacenters where sealers are deployed by looking at the proximities between their actual addresses and locations of the datacenters. Figure 1 summarizes the topological information we gathered from this inference.

9 Experiments

In this section, we present the double spending results of the Cloning Attack in both Aura and Clique. We first present our experimental setup then detail the risk for an attacker to perform double spending in both Aura and Clique.

9.1 Testnet setup

To practically observe the chance of successful double spending using the approaches described in the previous sections we have created our own PoA blockchain networks, experimented the attacks and measured their success rate empirically.

Our testnet consists of 10 Ubuntu 16.04 Virtual Machines (VMs) on our OpenStack private cloud; each VM is provided with 1 virtual CPU core and 2 GB of memory. These VMs are placed into two subnets, 5 VMs each; they are connected through 5 linux virtual routers and a physical Ethernet switch with dedicated VLAN. An instance of either Parity-Ethereum 2.0.8 stable version with Aura or go-ethereum 1.8.21 stable version with Clique runs on each VM.

All of these instances are peering with each other to form the blockchain network. While we have 10 Ethereum instances in total, our PoA blockchain employed only 9 unique private keys for sealers; the last instance instead uses the same key as the first one as explained in Section 5 where one instance is seen as a clone of the other.

In our experiments, the attacker is a Byzantine (or malicious) sealer with the intention to achieve double spending. This attacker is provided the capability to transiently partition the network into two sealer groups: the attacker and the victim group. We refer to the attacker group as the group of sealers whose blocks sealed during the network partition are intended to be adopted as a part of canonical chain, while we refer to the victim group as the group of sealers whose sealed blocks are intended to be discarded after the fork is resolved.

[7] https://blockscout.com/poa/core.
[8] https://core-netstat.poa.network/
[9] https://nmap.org/

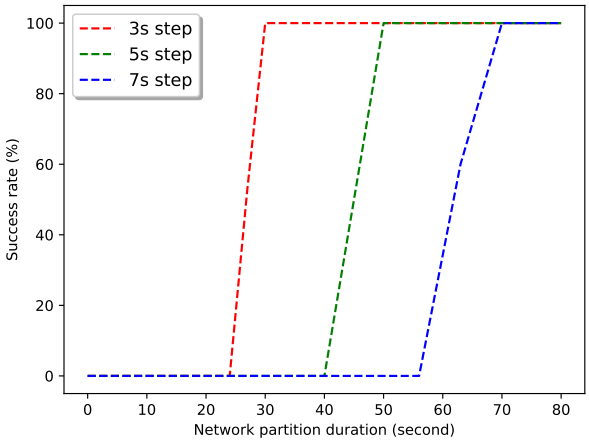

Figure 4: The success rate of double spending with the Cloning Attack in Aura

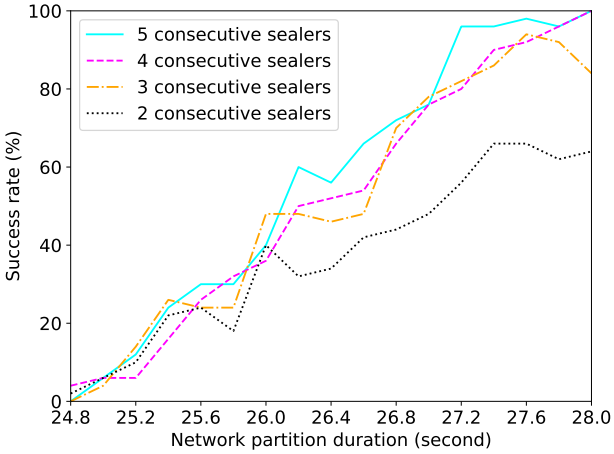

Figure 5: The success rate of double spending by network partition duration and number of consecutive possible in-order sealers in Clique

To grant the capability to partition the network, we allow our attacker to cut the network connectivity between two subnets using a firewall feature on the linux routers. Note that the same result is achieved using a man-in-the-middle attack though ARP-spoofing in a local area network or with BGP-hijacking in other networks [6]. The attacker is also provided the controllability over 2 Ethereum instances (2 VMs) that share the same private key used to seal the blocks.

The attacker aims to partition the network right before their turn to seal the block, where each sealer group must contain one VM that is under the control of the attacker. To begin the attack, our attacker actively checks the owner of the current turn every 10 ms in order to partition the network close to the right timing. Right after the network partition, the attacker issues one transaction to each sealer group; these two conflicting transactions TX_1 and TX_2, for example Alice is giving all of her coins to Bob in TX_1 and gives the same coins to Carol in TX_2.

After issuing the transactions, the network partition is maintained during a period that depends on which PoA algorithm as explained below. When the fork is resolved at the end of the network partition, we look at the resulting branch of the fork which has been adopted as a canonical chain as well as the status of transactions.

A double spending is considered successful only if:

1. A transaction issued to the victim group is committed before the end of the network partition;
2. the blocks sealed by the attacker group during the fork have been adopted as a part of the canonical chain after the end of the partition; and
3. the resulting canonical chain does not contain a transaction issued to the victim group.

9.2 Running the Cloning Attack against Aura

We experiment the Cloning Attack in Aura by varying the step duration and network partition duration. We chose Step durations 3, 5, and 7 seconds in order to observe their impact of the minimum partition duration that makes the attack successful.

We maintain the network partition to match the step duration in use, such that for example a 24, 27, and 30 second partition duration corresponds respectively to the 8th, 9th and 10th step for a 3 second step duration, respectively. We divide the sealers into two groups, such that apart from the two attacker instances, the placement of the reminder sealers is randomly but equally balanced between the two partitions. We do ensure, however, that both groups have an equal number of instances, which is 5, and each group contains one of two instances under the control of the attacker. The values plotted for each combination of step duration and network partition duration are the averages over 30 runs.

Figure 4 presents the double spending success rate of 3, 5 and 7 second step durations. In all three cases, the obtained results show a similar trend. As expected, achieving a successful double spending is impossible in all step durations at the 8th step or any earlier step, namely 24, 40 and 56 seconds for 3, 5, and 7 second step durations, respectively. Indeed, these attempts fail because any attack attempt among these runs could neither commit the transaction in the victim group nor force the block sealed by the attacker group to be adopted as a canonical chain when the network partition ends.

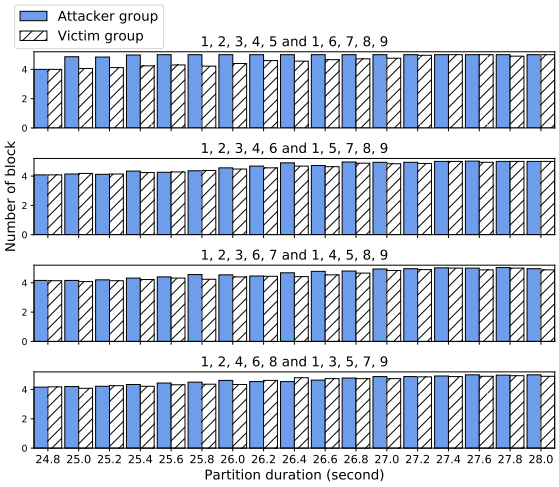

Figure 6: The average number of blocks created during various sealer partitions of different durations

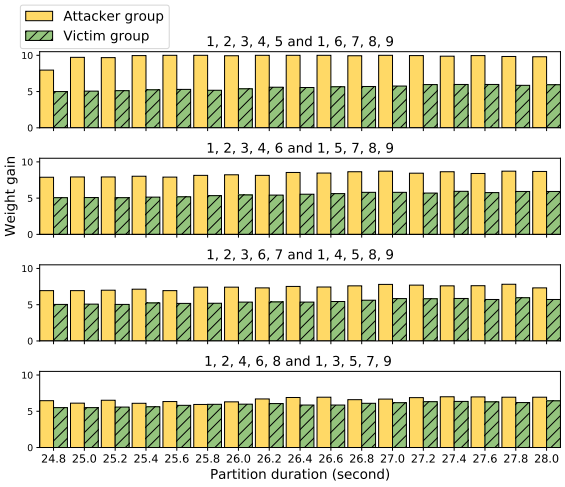

Figure 7: The average weight gained during various sealer partitions of different durations

However, we can observe that as expected, the chances of successful double spending at the 9th step falls within the range between 50-60%. Even though both groups are provided enough time to seal 5 blocks in order to commit the transactions, the attacker still cannot force a particular branch of the fork to be adopted. The variation at this point is due to the randomness of Ethereum instance placement during our experiment.

For all three step durations, at the 10th step and any step thereafter, the attack is always successful (100% chances). This is due to the attack technique in use that allows the attacker to force a branch of the fork to be adopted. Overall, we can see that a longer step duration requires a longer period of network partition in order to achieve a successful double spending, which confirms our expectations.

9.3 Running the Cloning Attack against Clique

We experiment the Cloning Attack on Clique while varying the partition duration and the way sealers are divided between two partitions.

The variations in the sealer divisions are included in the experiments in order to capture the changes in weight of each branch as a result of the sealing sequences.

In particular, we experiment with the 4 sealer divisions presented below with different number of consecutive sealers (note that these divisions allow the attacker group to seal different number of in-order blocks at the beginning of the partition):

- 5 consecutive sealers: 1, 2, 3, 4, 5 in the attacker group and 1, 6, 7, 8, 9 in the victim group;
- 4 consecutive sealers: 1, 2, 3, 4, 6 in the attacker group and 1, 5, 7, 8, 9 in the victim group;
- 3 consecutive sealers: 1, 2, 3, 6, 7 in the attacker group and 1, 4, 5, 8, 9 in the victim group;
- 2 consecutive sealers: 1, 2, 4, 6, 8 in the attacker group and 1, 3, 5, 7, 9 in the victim group.

The partition duration is based on the block duration in use, which is fixed to 5 seconds in all our Clique experiments. Since our testnet setup consists of 9 sealers in total, to commit a transaction during a partitioning, at least 5 blocks must be sealed in such a period. In the best case where 5 sealers could seal 5 in-order blocks, the minimum duration required for the attack to succeed is equal to $5 \times 5 = 25$ seconds. In other cases where at least 1 out of 5 blocks is sealed out-of-order, however, the required duration exceeds 25 seconds.

Based on our knowledge of the time necessary for the algorithm to seal 5 blocks, we vary the duration from 24.8 to 28.0 seconds in an incremental step of 200 milliseconds. The range of duration allows to take into account the random delay of out-of-order sealers as shown in Algorithm 3 and yet to capture the behavior of the system from the point where only 4 blocks can be sealed to the point where 5 blocks can be sealed.

For each run we keep a record of whether the double spending was successful, which block was sealed by which sealer, the weight gained during the partition for each fork, and the number of blocks created during the network partition. The values averaged over 50 runs are depicted in the charts for each partition duration.

Figure 5 reveals the double spending success rate for the four aforementioned sealer divisions while Figure 6 and Fig-

ure 7 show the number of sealed blocks and the weight gained during the network partition, respectively. We observe that the success rate in Figure 5, follows a similar trend for all of 4 grouping variations; the longer the partition duration, the higher the chance of successful double spending.

The shortest partition duration value in the chart, 24.8 second, gives the lowest success rate regardless of the sealer division. This low success rate for short duration can be explained by the number of blocks sealed during the network partition. Indeed, due to the limited partition duration, the victim group is rarely able to seal five blocks during the network partition as shown in Figure 6, thus a transaction issued by the victim group could not be committed and the attack fails. When the partition takes longer, we can see that the victim group is able to seal five blocks.

When the partition duration is less or equal to 26 seconds, there is no noticeable difference between the four different sealer divisions. In the case of two consecutive possible in-order sealers and when the partition duration is greater than 26 seconds, however, the success rate is lower than the other three divisions. This phenomenon can be explained by the weight gained during the network partition as shown in Figure 7. In fact, in case of 2 consecutive possible in-order sealers, the difference in weight gained between attacker and victim branches becomes relatively low; this gap narrows with the increase in the partition duration.

10 Discussion and Countermeasures

In this section, we compare the vulnerabilities of Aura and Clique to the Cloning Attack resulting from our experiments in Section 9. We then present potential countermeasures against the Cloning Attacks and discuss their implication to the blockchain safety and liveness.

10.1 Comparison between Aura and Clique

In this section, we explain why the Cloning Attack against Aura can always be successful whereas the Cloning attack against Clique is much faster but not always successful.

As detailed in Section 4, one of the main differences between Aura and Clique resides in the predictability of the sequence of sealers. In fact, in Aura the sequence is strictly enforced whereas in Clique this sequence may change depending on the difference between a random number and the network communication delay. This slight algorithmic difference has however significant consequences on consensus algorithms resilience to double spending attacks using our proposed Cloning Attacks.

On the one hand and as we have demonstrated in Section 9, due to its strict enforcement of sealing order, Aura is vulnerable to the Cloning attack in case of network partition. Performing the Cloning attack against Aura, the attacker does not need to know anything about the identity of the sealers

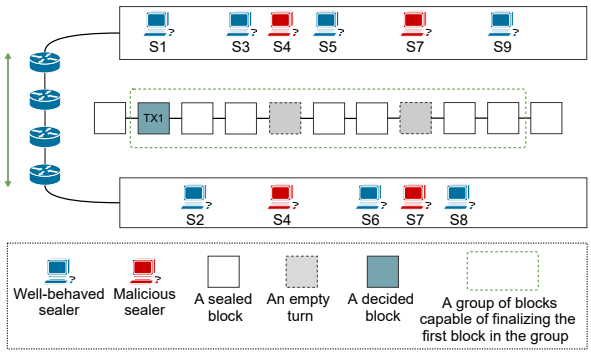

Figure 8: Modified Aura using two-third consensus with 7 well-behaved and 2 malicious sealers

nor does it need to know their order. Thus, a malicious sealer only needs to partition the overlay network using classical network attacks such as BGP hijacking to succeed in double spending with a 100% chance of success.

On the other hand, double spending without topology information on Clique is possible, but the attack against Clique is about twice as fast as against Aura when the topology is known. Indeed, as we have presented in Section 9, the knowledge of potential next in-order sealer greatly influences the chance of double spending. When the attacker is capable of isolating the next $\lfloor n/2 \rfloor + 1$ sealers, it is able to perform the double spending attack with 100% success rate. On the opposite, the knowledge of only the next two in-order sealers only guarantees a success rate of 60% maximum.

Interestingly, when considering the attacks against both Aura and Clique without the knowledge of the topology, it appears that attacking Clique can be even slower than attacking Aura. The reason is that in the worst case scenario where all in-order sealers are on the victim side, the attacker will have to obtain a branch that is twice as large as the victim branch before it can double spend. Growing this branch would take more time than executing the Cloning attack on Aura. But overall, even without knowledge of the topology both Aura and Clique consensus algorithms are vulnerable to a malicious sealer aiming at double spending.

10.2 Countermeasures to the Cloning Attack

In this section, we introduce the countermeasures that could be employed to totally prevent the attack or mitigate its risk.

10.2.1 Partially synchronous consensus algorithms

Instead of relying on the PoA consensus that suffers from unexpected message delays, one could use a deterministic consensus algorithm that is partially synchronous in that it

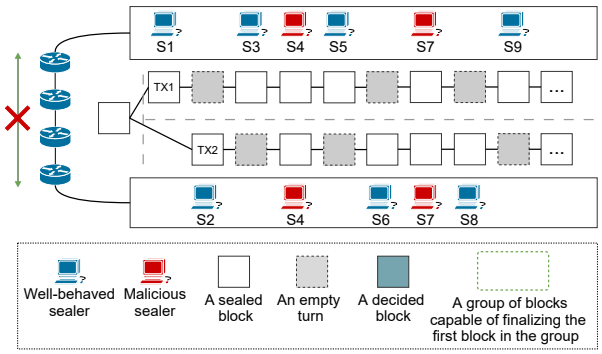

Figure 9: Modified Aura using two-third consensus with 7 well-behaved and 2 malicious sealers during a network partition

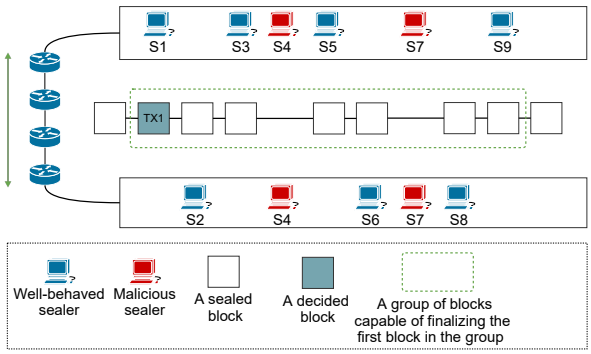

Figure 10: Modified Clique using two-third consensus with 7 well-behaved and 2 malicious sealers

tolerates arbitrary delays. PBFT [2] is one example as it relies on a leader and was not designed to scale outside a small network. DBFT [3] is another deterministic partially synchronous consensus algorithm but was especially designed for blockchain systems. DBFT does not rely on a correct leader to make progress, and was designed to scale to large number of participants. In addition, DBFT is time optimal and resilience optimal. It has been shown that DBFT is resilient to double spending attacks, as it is not possible for a blockchain building upon it, like the Red Belly Blockchain [19], to fork.

10.2.2 Requiring two-thirds of sealers

Based on our understanding of how the two attacks against PoA work, we believe that the two PoA algorithms can sacrifice blockchain liveness during network partition in exchange for safety if they consider block finality only when strictly more than two-thirds of sealers have sealed blocks on the chain. As long as the number of malicious sealers is less than one-third of the total sealers, the modified algorithms will be able to guarantee safety, and the liveness will still be preserved whenever the network is not partitioned.

Figure 8 and Figure 9 demonstrate how a modified version of Aura with two-thirds sealers works both with and without a partition in case of a blockchain network with 9 sealers: S1 to S9. Since the maximum number of tolerable malicious sealers must be less than one-third of the total 9 sealers, we include 2 malicious sealers in the setup: S4 and S7; these two sealers are allowed to be silent or even seal the blocks on all the partitions whenever a network partition occurs. To finalize a block that contains a transaction, the blockchain network needs strictly more than two-thirds of 9 sealers to seal the blocks on the chain; in other words, at least 7 sealers are necessary in order to decide a block and commit a transaction.

As indicated with a green dash frame in Figure 8, this version of Aura can commit TX_1 even though S4 and S7 do not contribute any block to the chain in their turns. The S4 turn can be left empty and S5 may simply continue sealing a block after that; the same goes for S8 after the end of S7 turn. In this case, the other 7 honest sealers alone are sufficient to finalize a block that contains TX_1, because 7 is greater than two-thirds of $n = 9$.

The Figure 9 illustrates a case where the network partition occurs and the two malicious sealers S4 and S7 are capable of sealing the blocks in both groups simultaneously. Although it is possible to partition in such a way that one group has at least 7 sealers, it is impossible to have at least 7 sealers on both sides at the same time. Therefore, it is impossible to commit the transactions issued to both partitions concurrently, even though Aura allows their sealers to continue sealing more blocks during the network partition.

The same condition also applies to Clique in a similar manner. To illustrate how a modified Clique with two-thirds consensus works, let us use the same setup with a blockchain network of $n = 9$ sealers: S1 to S9 with two malicious sealers S4 and S7. As indicated with green dash frame in Figure 10, the blockchain network can still commit TX_1. Instead of S4 and S7, Clique allows other sealers to continue sealing the blocks in an out-of-order fashion without any gap and eventually reaches the 7 blocks requirement. In another case where the network partition occurs, if the number of sealers in a partition is not greater than two-thirds of all sealers, such a partition will be stuck even before sealing enough blocks to commit a transaction. Figure 11 shows a concrete example where both partitions get stuck due to the lack of sealers; one partitions contains only 6 sealers where there are only 5 sealers in the other.

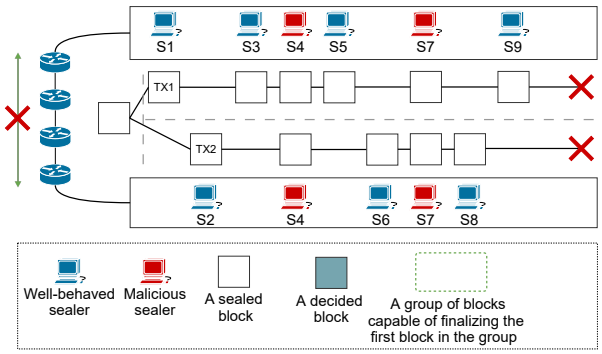

Figure 11: Modified Clique using two-third consensus with 7 well-behaved and 2 malicious sealers during a network partition

10.2.3 Adopting a longer step duration in Aura or a longer block period in Clique

Adopting a longer required time to create one block forces the attacker to maintain the network partition for a longer period. In general, a long duration of network partition attack is more difficult to achieve than a short one for two reasons. First, there might be a regular change in the network. Second such an attack might be detected and an action can be taken as a response to remedy the lack of connectivity.

On the downside, adopting this countermeasure lowers the throughput of the blockchain. As blocks would require a longer duration to be decided, this approach increases the time needed for transactions to be committed or final.

11 Conclusion

Blockchains based on *proof-of-work* protocols typically allow distinct blocks to be appended to the same index of the chain and these forks lead to security vulnerabilities when growing without being noticed. As a result, Byzantine fault tolerance has been introduced in the most popular implementations of the Ethereum protocol as *proof-of-authority*. As the industry is building upon these protocols to use Ethereum in a consortium of institutions, it has become crucial to assess their vulnerability.

In this paper, we have presented the Cloning Attack against these protocols that leads to double spending. First, we explained how to gather the necessary topological information to identify how to attack the underlying network of these blockchains. Second and more importantly, we showed that the cloning of two Ethereum instances is sufficient to steal assets in these services through double spending. This lack of safety is concerning for the numerous critical applications that aim at tracking ownership of valuable assets.

Finally, we present a simple counter measure that remedies this safety violation by blocking the service in the worst case but always preventing an attacker from stealing assets.

Acknowledgements

This research is in part supported under Australian Research Council Discovery Projects funding scheme (project number 180104030) entitled “Taipan: A Blockchain with Democratic Consensus and Validated Contracts”. Vincent Gramoli is a Future Fellow of the Australian Research Council.

References

[1] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. In *Proceedings of the Second Italian Conference on Cyber Security, 2018*, 2018.

[2] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI ’99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[3] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*, pages 1–8, 2018.

[4] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, pages 198–208, 1983.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery, Vol. 35, No. 2, pp.288-323*, 1988.

[6] Parinya Ekparinya, Vincent Gramoli, and Guillaume Jourjon. Impact of man-in-the-middle attacks on ethereum. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2018.

[7] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proc. of FC 2014*, pages 436–454, 2014.

[8] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. of*

the 26th Symposium on Operating Systems Principles, pages 51–68, 2017.

[9] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security*, pages 129–144, 2015.

[10] Pavel Khahulin Igor Barinov, Viktor Baranov. POA network white paper, Sept. 2018. https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper.

[11] Jae Kwon. Tendermint, consensus without mining, 2015. Unpublished manuscript.

[12] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[13] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smartpool: Practical decentralized pooled mining. In *USENIX Security*, pages 1409–1426, 2017.

[14] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS'16*, pages 31–42, New York, NY, USA, 2016. ACM.

[15] C. Natoli and V. Gramoli. The Blockchain Anomaly. In *2016 IEEE NCA*, pages 310–317, October 2016.

[16] Christopher Natoli and Vincent Gramoli. The balance attack or why forkable blockchains are ill-suited for consortium. In *IEEE/IFIP DSN'17*, Jun 2017.

[17] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *Proc. of IEEE EuroS&P 2016*, pages 305–320, 2016.

[18] Meni Rosenfeld. Analysis of hashrate-based double-spending, 2012.

[19] Vincent Gramoli Tyler Crain, Christopher Natoli. Evaluating the red belly blockchain. Technical Report 1812.11747, arXiv, 2018.

[20] Ingo Weber, Qinghua Lu, An Binh Tran, Amit Deshmukh, Marek Gorski, and Markus Strazds. A platform architecture for multi-tenant blockchain-based systems. In *IEEE International Conference on Software Architecture (ICSA)*, 2019.