Denial-of-Service Vulnerability of Hash-based Transaction Sharding: Attack and Countermeasure

Truc Nguyen and My T. Thai

Abstract—Since 2016, sharding has become an auspicious solution to tackle the scalability issue in legacy blockchain systems. Despite its potential to strongly boost the blockchain throughput, sharding comes with its own security issues. To ease the process of deciding which shard to place transactions in, existing sharding protocols use a hash-based transaction sharding in which the hash value of a transaction determines its output shard. Unfortunately, we show that this mechanism opens up a loophole that could be exploited to conduct a single-shard flooding attack, a type of Denial-of-Service (DoS) attack, to overwhelm a single shard that ends up reducing the performance of the system as a whole.

To counter the single-shard flooding attack, we propose a countermeasure that essentially eliminates the loophole by rejecting the use of hash-based transaction sharding. The countermeasure leverages the Trusted Execution Environment (TEE) to let blockchain's validators securely execute a transaction sharding algorithm with a negligible overhead. We provide a formal specification for the countermeasure and analyze its security properties in the Universal Composability (UC) framework. Finally, a proof-of-concept is developed to demonstrate the feasibility and practicality of our solution.

Index Terms—Blockchain, denial-of-service, trusted execution environment, flooding

1 Introduction

Sharding, an auspicious solution to tackle the scalability issue of blockchain, has become one of the most trending research topics and been intensively studied in recent years [1], [2], [3], [4], [5], [6], [7], [8], [9]. In the context of blockchain, sharding is the approach of partitioning the set of nodes (or validators) into multiple smaller groups of nodes, called shard, that operate in parallel on disjoint sets of transactions and maintain disjoint ledgers. By parallelizing the consensus work and storage, sharding reduces drastically the storage, computation, and communication costs that are placed on a single node, thereby scaling the system throughput proportionally to the number of shards. Previous studies [2], [3], [5] show that sharding could potentially improve blockchain's throughput to thousands of transactions per second (whereas the current Bitcoin system only handles up to 7 transactions per second and requires 60 minutes confirmation time for each transaction).

Despite the incredible results in improving the scalability, blockchain sharding is still vulnerable to some severe security problems. The root of those problems is that, with partitioning, the honest majority of mining power or stake share is dispersed into individual shards. This significantly reduces the size of the honest majority in each shard, which in turn dramatically lowers the attack bar on a specific shard. Hence, a blockchain sharding system must have some mechanisms to prevent adversaries from gaining the majority of validators of a single shard, this is commonly referred to as single-shard takeover attack.

• *T. Nguyen and My T. Thai are with the Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL, 32611.*
E-mail: truc.nguyen@ufl.edu and mythai@cise.ufl.edu

In this paper, we take a novel approach by exploiting the inter-shard consensus to identify a new vulnerability of blockchain sharding. One intrinsic attribute of blockchain sharding is the existence of cross-shard transactions that, simply speaking, are transactions that involve multiple shards. These transactions require the involved shards to perform an inter-shard consensus mechanism to confirm the validity. Hence, intuitively, if we could perform a Denial-of-Server (DoS) attack to one shard, it would also affect the performance of other shards via the cross-shard transactions. Furthermore, both theoretical and empirical analysis [2], [5] show that most existing sharding protocols have 99% cross-shard transactions. This implies that an attack on one shard could potentially impact the performance of the entire blockchain. In addition, with this type of attacks, the attacker is a client of the blockchain system, hence, this attack can be conducted even when we can guarantee the honest majority in every shard.

Although existing work does have some variants of flooding attacks that try to overwhelm the entire blockchain by having the attacker generate a superfluous amount of dust transactions [10], [11], it is unclear how we could conduct this attack in a sharding system. In fact, we emphasize that a conventional transactions flooding attack on the entire blockchain (as opposed to a single shard) would not be effective for two reasons. First, blockchain sharding has high throughput, hence, the cost of attack would be enormous to generate a huge amount of dust transactions that is sufficiently much greater than the system throughput. Second, more importantly, since the sharding system scales with the number of shards, it can easily tolerate such attacks by adding more shards to increase throughput.

To bridge this gap, we propose a single-shard flood-

ing attack to exploit the DoS vulnerability of blockchain sharding. Instead of overwhelming the entire blockchain, an attacker would strategically place a tremendous amount of transactions into one single shard in order to reduce the performance of that shard, as the throughput of one shard is not scalable. The essence of our attack comes from the fact that most sharding proposals use hash-based transaction sharding [1], [2], [3], [6]: a transaction's hash value is used to determine which shard to place the transaction (i.e., output shard). Since that hash value (e.g., SHA-256) of a transaction is indistinguishable from that of a random function, this mechanism can efficiently distribute the transactions evenly among the shards and thus widely adopted. Therefore, an attacker can manipulate the hash to generate an excessive amount of transactions to one shard.

As we argue that using hash values to determine the output shard is not secure, we propose a countermeasure to efficiently eliminate the attack for the sharding system. By not using the transaction's hash value or any other attributes of the transaction, we can delegate the task of determining the output shard to the validators, then the adversary cannot carry out this DoS attack. However, this raises two main challenges: (1) what basis can be used to determine the output shard of a transaction, and (2) how honest validators can agree on the output of (1). For the first challenge, we need a *transaction sharding algorithm* to decide the output shard for each transaction. OptChain [5] is an example algorithm where it aims to minimize the number of cross-shard transactions and also balance the load among the shards. For the second challenge, a naive solution is to have the validators reach on-chain consensus on the output shard of every transaction. However, that would be very costly and reject the main concept of sharding, that is, each validator only processes a subset of transactions to parallelize the consensus work and storage.

To overcome the aforementioned challenge, we establish a system for executing the transaction sharding algorithm off-chain and attesting the correctness of the execution. As blockchain validators are untrusted, we need to guarantee that the execution of the transaction sharding algorithm is tamper-proof. To accomplish this, we leverage the Trusted Execution Environment (TEE) to isolate the execution of the algorithm inside a TEE module, shielding it from potentially malicious hosts. With this approach, we are not imposing any significant on-chain computation overhead as compared to the hash-based transaction sharding and also maintain the security properties of the blockchain. Moreover, this solution can be easily integrated into existing blockchain sharding proposals, and as modern Intel CPUs from 2014 support TEE, the proposed countermeasure is compatible with current blockchain systems.

Contribution. Our main contributions are as follows:

- We identify a new attack on blockchain sharding that exploits the loophole of using hash-based transaction sharing, namely single-shard flooding attack.
- To evaluate the potential impact of this attack on the blockchain system, we develop a discrete-event simulator for blockchain sharding that can be used to observe how sharding performance changes when the system is under attack. Not only for our attack analysis purposes, this simulator can also assist the research community in evaluating the performance of a sharding system without having to set up multiple computing nodes.
- We propose a countermeasure to the single-shard flooding attack by executing transaction sharding algorithms using TEE. Specifically, we provide a formal specification of the system and formally analyze its security properties in the Universal Composability (UC) framework with a strong adversarial model.
- To validate our proposed countermeasure, we develop a proof-of-concept implementation of the system and provide a performance analysis to demonstrate its feasibility.

Organization. The rest of the paper is structured as follows. Some background and related work are summarized in Section 2. Section 3 describes in detail the single-shard flooding attack with some preliminary analysis to demonstrate its practicality. In Section 4, we present the construction of our simulator and conduct some experiments to demonstrate the damage of the attack. The countermeasure is discussed in Section 5 with a formal specification of the system. Section 6 gives a security analysis of the countermeasure along with a performance evaluation on the proof-of-concept implementation. Finally, Section 7 concludes our paper.

2 Background and Related Work

Blockchain sharding. Several solutions [1], [2], [3], [4], [5], [6], [7], [8], [9] suggest partitioning the blockchain into shards to address the scalability issue in Bitcoin blockchain. Typically, with sharding, the blockchain's state is divided into multiple shards, each has its own independent state and transactions and is managed by the shard's validators. By having multiple shards where each of them processes a disjoint set of transactions, the computation power is parallelized, and sharding in turn helps boost the system throughput with respect to the number of shards. Some main challenges of a sharding protocol include (1) how to securely assign validators to shards, (2) intra-shard consensus, (3) assigning transactions to shards, and (4) processing cross-shard transactions. Our proposed attack exploits the third and fourth challenges of sharding that deal with transactions in a sharding system.

In a simple manner, a transaction is cross-shard if it requires confirmations from more than one shard. In the Unspent Transaction Outputs (UTXO) model used by Bitcoin, each transaction has multiple outputs and inputs where an output dictates the amount of money that is sent to a Bitcoin address. Each of the outputs can be used as an input to another transaction. To prevent double-spending, an output can be used only once. Denote tx as a transaction with two inputs tx_1 and tx_2, this means tx uses one or more outputs from transaction tx_1 and tx_2. Let S_1, S_2, and S_3 be the shards containing tx_1, tx_2, and tx, respectively, we refer S_1 and S_2 as the *input shards* of tx, and S_3 as the *output shard*. If these three shards are the same, tx is an in-shard transaction, otherwise tx is cross-shard.

To determine the output shard of a transaction, most sharding protocols use the hash value of the transaction to calculate the ID of the output shard. By leveraging the hash value, the transactions are effectively assigned to shards in

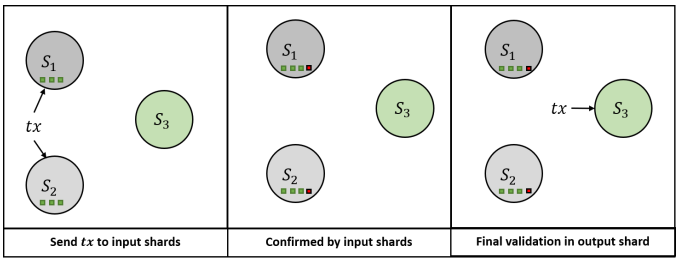


Fig. 1: Processing cross-shard transaction tx. tx has input shards S_1, S_2, and output shard S_3. tx has to wait to be confirmed in S_1, S_2 before it can be validated in S_3.

a uniformly random manner. However, we show that this mechanism can be manipulated to perform a DoS attack.

To process cross-shard transactions, several cross-shard validation mechanisms have been proposed [2], [3], [4]. A cross-shard validation mechanism determines how input and output shards can coordinate to validate a cross-shard transaction. This makes the process of validating cross-shard transactions particularly expensive since the transaction must wait for confirmations from all of its input shards before it can be validated in the output shard. Fig. 1 illustrates this process. Our attack takes advantage of this mechanism to cause a cascading effect that creates a negative impact on shards that are not being attacked.

Denial-of-service and flooding attacks. Denial-of-service (DoS) is commonly defined as an intentional attack on availability and it has been around for decades [12]. In general, it is hard to defend against the DoS attacks and we typically can only mitigate them. A common mitigation technique is based on anomaly detection to filter out DoS attack packets [13], [14], [15].

In the context of Bitcoin, DoS typically takes the form of a flooding attack that overwhelms the system with a flood of transactions. Over the years, we have observed the economic impact of this attack as Bitcoin has been flooded with dust transactions by malicious users to make legitimate users pay higher mining fees [16]. There exist some variants of flooding attack that aim to overwhelm an entire blockchain system [10], [11], not a single shard. The main concept of the attack is to send a huge amount of transactions to overwhelm the mempool, fill blocks to their maximum size, and effectively delay other transactions. Typically, unconfirmed transactions are stored in the mempools managed by blockchain validators. In contrast to the limited block size, the mempool size has no size limit.

This kind of attack requires the attacker to flood the blockchain system at a rate that is much greater than the system throughput. Intuitively, such an attack is not effective on a sharding system because its throughput is exceedingly high. In this paper, we show how attackers can manipulate the transaction's hash to overwhelm a single shard, thereby damaging the entire blockchain through cascading effects caused by cross-shard transactions.

Blockchain on Trusted Execution Environment (TEE). A key building block of our countermeasure is TEE. Memory regions in TEE are transparently encrypted and integrity-protected with keys that are only available to the processor. TEE's memory is also isolated by the CPU hardware from the rest of the host's system, including high-privilege system software. Thus the operating system, hypervisor, and other users cannot access the TEE's memory. Among available implementations of TEE, Intel SGX [17] supports generating remote attestations that are used to prove the correct execution of programs running inside TEE.

There has been a recent growth in adopting TEEs to improve blockchains [18], [19], [20], [21], but not sharding systems. Teechain [19] proposes an improvement over the off-chain payment network in Bitcoin using TEE to enable asynchronous blockchain access. BITE [18] leverages TEE to further enhance the privacy of Bitcoin's clients. In [20], [21], the authors develop secure and efficient smart contract platforms on Bitcoin and Ethereum, respectively, using TEE as a module to execute the contract's code.

We argue that TEE can be used to develop an efficient countermeasure for the single-shard flooding attack in which transaction sharding algorithms can be securely executed inside a TEE module. However, since existing solutions are designed to address some very specific issues such as smart contracts or payment networks, applying them to blockchain *sharding* systems is not straightforward.

3 Single-shard flooding attack

In this section, we describe our proposed single-shard flooding attack on blockchain sharding starting with the threat model and detail on performing the attack. Then, we present some preliminary analysis of the attack to illustrate its potential impact and practicality.

3.1 Threat Model

Attacker. We use Bitcoin-based sharding systems, such as OmniLedger, RapidChain, and Elastico, as the attacker's target. We consider an attacker who is a client of the blockchain system such that:

1) The attacker possesses enough amount of Bitcoin addresses to perform the attack. In practice, Bitcoin addresses can be generated at no cost.
2) The attacker has spendable bitcoins in its wallet and the balance is large enough to issue multiple transactions between its addresses for this attack. Each of the transactions is able to pay the minimum relay fee $minRelayTxFee$. We will discuss the detailed cost in the next section.
3) The attacker is equipped with software that is capable of generating transactions at a rate that is higher than a shard's throughput, which will be discussed in the subsequent section.
4) Since this is a type of DoS attack, to prevent it from being blocked by the blockchain network, the attacker can originate the attack from multiple sources. The attacker can also leverage some kind of anonymous communication when connecting to the Bitcoin network to prevent the network packets from revealing the attacker's identity. Therefore, we assume that the attacker can remain anonymous and untraceable.

Goals of attacks. By employing the concept of flooding attack, the main goal of the single-shard flooding attack is to overwhelm a single shard by sending a huge amount

of transactions to that shard. The impact of this attack has been widely studied on non-sharded blockchains like Bitcoin such that it can reduce the system performance by delaying the verification of legitimate transactions and eventually increase the transaction fee.

Furthermore, with the concept of cross-shard transactions where each transaction requires confirmation from multiple shards, an attack to overwhelm one shard could affect the performance of other shards and reduce the system's performance as a whole. For example, in Fig. 1, if S_1 were under attack, the transaction validation would also be delayed in S_3. Previous work [2] shows that placing transactions using their hash value could result in 99.98% of cross-shard transactions. Since the throughput of one shard is limited, under our attack, it could effectively become the performance bottleneck of the whole system. Therefore, to make the most out of this scheme, the attacker would target the shard that has the lowest throughput in the system.

How to perform attack. In most Bitcoin-based sharding systems, such as OmniLedger, RapidChain, and Elastico, the hash of a transaction determines which shard to put the transaction in. Specifically, the ending bits of the hash value indicate the output shard ID. The main idea of our attack is to have the attackers manipulate the transaction's hash in order to place it into the shard that they want to overwhelm. To accomplish this, we conduct a brute-force generation of transactions by alternating the output addresses of a transaction until we find an appropriate hash value.

Let T be the shard that the attacker wants to overwhelm. We define a "malicious transaction" as a transaction whose hash was manipulated to be put in shard T. Denote tx as a transaction, $tx.in$ is the set of input addresses, and $tx.out$ is the set of output addresses. We also denote O as the set of attacker's addresses, $I \subseteq O$ as the set of attacker's addresses that are holding some bitcoins. Let $\mathcal{H}(\cdot)$ be the SHA-256 hash function (its output is indistinguishable from that of a random function), Algorithm 1 describes how to generate a malicious transaction in a system of 2^N shards.

Starting with a raw transaction tx, the algorithm randomly samples a set of input addresses for $tx.in$ from I such that the balance of those addresses is greater than the minimum relay fee. It then randomly samples a set of output addresses for $tx.out$ from O and set the values for $tx.out$ so that tx can pay the minimum relay fee. The hash value of the transaction is determined by double hashing the transaction's data using the SHA-256 function. It checks if the final N bits indicate T ($\&$ denotes a bitwise AND), if that is true, it outputs the malicious tx that will be placed into shard T. Otherwise, it re-samples another set of output addresses for $tx.out$ from O.

Algorithm 1 Generate a malicious transaction

Input: $\mathcal{I}, \mathcal{O}, N, T$
Output: A malicious transaction tx

1: $tx \leftarrow$ raw transaction
2: $tx.in \xleftarrow{\$} \mathcal{I}$
3: **while** $\mathcal{H}(\mathcal{H}(tx))$ & $(1^{256-N} \parallel 0^N) \neq T$ **do**
4: $\quad tx.out \xleftarrow{\$} \mathcal{O}$
5: $\quad$ Set values for $tx.out$ to satisfy the $minRelayTxFee$.
6: **end while**
7: **Ret** tx

TABLE 1: Capability of generating malicious tx with respect to the no. of shards on an Intel Core i7 laptop with 8 threads.

No. of shards	No. of malicious tx per sec
2	823,512
4	412,543
8	205,978
16	103,246
32	52,361
64	26,939

3.2 Preliminary Analysis

3.2.1 Capability of generating malicious transactions

In this section, we demonstrate the practicality of the attack by assessing the capability of generating malicious transactions on a real machine. Suppose we have 2^N shards and a transaction tx, that means we will use N ending bits of $\mathcal{H}(tx)$ to determine its shard. Suppose we want to put all transactions into shard 0, we need to generate some malicious transactions tx such that the last N bits of $\mathcal{H}(tx)$ must be 0. We calculate the probability of generating a malicious transaction as follows. As a SHA-256 hash has 256 bits, the probability of generating a hash with N ending zero bits will be $\frac{2^{256-N}}{2^{256}} = \frac{1}{2^N}$. Therefore, we expect to obtain 1 malicious transaction per generating 2^N transactions. That means if we have 16 shards, we can obtain 1 malicious tx (i.e., the last 4 bits are zero) per generating 16 transactions.

To see the capability of generating malicious transactions, we conduct an experiment on an 8th generation Intel Core i7 laptop. The program to generate transactions is written in C++ and runs with 8 threads. When the number of shards is 64, the program can generate up to 1,644,736 transaction hashes per second, of which there are 26,939 malicious transactions (8 ending bits are zero). In short, within 1 second, a laptop can generate about 26,939 malicious transactions, which is potentially much more than the throughput of one shard. Table 1 shows the number of malicious transactions generated per second with respect to the number of shards. Note that, in practice, an attacker can easily produce much higher numbers by using a more highly capable machine with a faster CPU.

3.2.2 Cost of attacks

The default value of $minRelayTxFee$ in Bitcoin is 1,000 satoshi per kB, which is about $0.10 (as of Feb 2020). Taking into account that the average transaction size is 500 bytes, each transaction needs to pay $0.05 as the $minRelayTxFee$. Our experiments below show that generating 2,500 malicious transactions is enough to limit the throughput of the whole system by that of the attacked shard. Hence, the attacker needs about $125 to perform the attack effectively. Furthermore, by paying the minimum relay fee without paying the minimum transaction fee, the malicious transactions will still be relayed to the attacked shard's mempool but will not be confirmed, thereby retaining the starting balance.

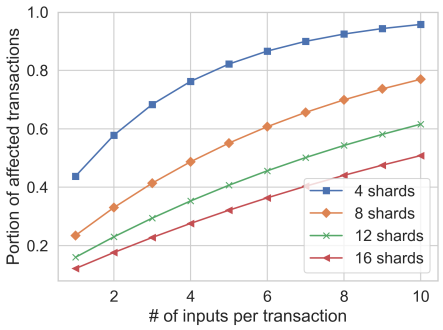


Fig. 2: Affected transactions by the single-shard flooding attack

3.2.3 Cascading effect of the single-shard flooding attack

We estimate the portion of transactions that are affected by the attack. A transaction is affected if one of its input shards or the output shard is the attacked shard. In [9], the authors calculate the ratio of transactions that could be affected when one shard is under attack. Specifically, when considering a system with n shards and transactions with m inputs, the probability of a transaction to be affected by the attack is $1 - (\frac{n-1}{n})^{m+1}$. The result is illustrated in Fig. 2. As can be seen, a typical transaction with 2 or 3 inputs has up to 70% chance of being affected with 4 shards. However, with 16 shards about 20% of the transactions are still affected. Note that this number only represents the transactions that are "directly" affected by the attack, the actual number is higher when considering transactions that depend on the delayed transactions.

Even though the analysis shows that the number of affected transactions is less at 16 shards than at 4 shards, in fact, the attack does much more damage to the 16-shard system. The intuition of this scenario is that as we increase the number of shards, we also increase the number of input shards per transaction. Since a transaction has to wait for the confirmations from all of its input shards, an affected transaction in the 16-shard system takes more time to be validated than that in the 4-shard system. The experiments in the next section will illustrate this impact in more detail.

4 Analyzing the Attack's Impacts

In this section, we present a detailed analysis of our attack, especially how it impacts the system performance as a whole. Before that, we describe the design of our simulator that is developed to analyze the performance of a blockchain sharding system.

4.1 Simulator

Our implementation was based on SimBlock [22], a discrete-event Bitcoin simulator that was designed to test the performance of the Bitcoin network. SimBlock is able to simulate the geographical distribution of Bitcoin nodes across six regions (North America, South America, Europe, Asia, Japan, Australia) of which the bandwidth and propagation delay are set to reproduce the actual behavior of the Bitcoin network. Nevertheless, SimBlock fails to capture the role of

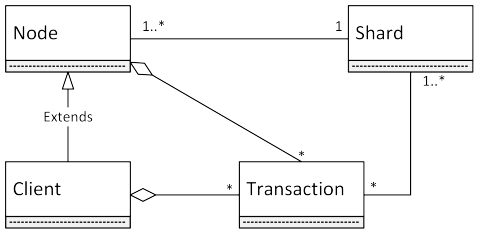


Fig. 3: UML class diagram of the simulator

transactions in the simulation, which is an essential part in evaluating the performance of blockchain sharding systems.

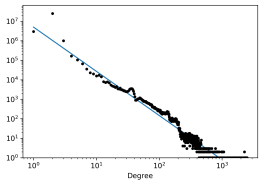
Our work improves SimBlock by taking into consideration the Bitcoin transactions and simulating the behavior of sharding. Fig. 3 shows a simple UML class diagram depicting the relations between components of our simulator. As can be seen later, our simulator can be easily used to evaluate the performance of any existing or future sharding protocols.

Transactions. The sole purpose of SimBlock was only to show the block propagation so the authors did not consider transactions. To represent Bitcoin transactions, we adopt the Transaction-as-Nodes (TaN) network proposed in [5]. Each transaction is abstracted as a node in the TaN network, there is a directed edge (u, v) if transaction u uses transaction v as an input. In our simulator, each transaction is an instance of a Transaction class and can be directly obtained from the Bitcoin dataset. At the beginning of the simulation, a Client instance loads each transaction from the dataset and sends them to the network to be confirmed by Nodes. Depending on the transaction sharding algorithm, each transaction could be associated with one or more shards.

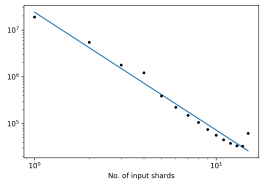
Furthermore, our simulator can also emulate real Bitcoin transactions in case we need more transactions than what we have in the dataset or we want to test the system with a different set of transactions. With regard to sharding, the two important factors of a transaction are the degree and the input shards. From the Bitcoin dataset of more than 300 million real Bitcoin transactions, we fit the degree distribution with a power-law function as in Fig. 4a (black dots are the data, and the blue line is the resulting power-law function). The resulting function is $y = 10^{6.7}x^{-2.3}$. Fig. 4b shows the number of input shards with 16 shards (using hash-based transaction sharding) that is also fitted with a power-law function. The resulting function is $y = 10^{7.2}x^{-2.2}$. Hence, the Client can use these distributions to sample the degree and input shards when generating transactions that resemble the distribution of the real dataset.

Sharding. After the simulator generates Node instances, each of them is distributed into an instance of Shard. All nodes in a shard share the same ledger and a mempool of unconfirmed/pending transactions. In the class Node, we implement a cross-shard validation algorithm that decides how nodes in different shards can communicate and confirm cross-shard transactions. In the current implementation, we use the mechanism proposed in [3] to process cross-shard transactions.

For each Node instance, upon receiving a transaction, it will relay the transaction to the destination shard. When the transaction reaches the shard, it will be stored in the

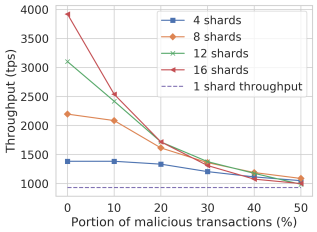


(a) Degree distribution of the Bitcoin transactions. Fitted power-law $y = 10^{6.7}x^{-2.3}$.

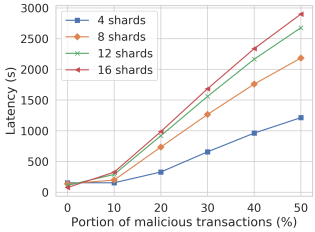


(b) Distribution of Bitcoin transactions' number of input shards. Fitted power-law $y = 10^{7.2}x^{-2.2}$.

Fig. 4: Bitcoin transactions. The black dots are the data, the blue line shows a fitted power-law function



(a) Impact on system throughput



(b) Impact on system latency

Fig. 5: Impact on system throughput and latency

mempool of the Node instances in that shard. Each transaction in the mempool is then validated using an intra-shard consensus protocol.

Use cases of the simulator. Besides being used to test the impact of our proposed attack, researchers can also use the simulator to evaluate the performance of multiple blockchain sharding systems. By far, most experiments on blockchain sharding have to be run on numerous rented virtual machines [1], [2], [3], [4], this is notably costly and complicated to set up. Without having to build the whole blockchain system, our simulator is particularly useful when researchers need to test various algorithms and system configurations on blockchain long before deploying the real system.

By using simulation, various setups can be easily evaluated and compared, thereby making it possible to recognize and resolve problems without the need of performing potentially expensive field tests. By exploiting a pluggable design, the simulator can be easily reconfigured to work with different algorithms on transaction sharding, cross-shard validation, validators assignment, and intra-shard consensus protocol.

4.2 Experimental Evaluations

Our experiments are conducted on 10 million real Bitcoin transactions by injecting them into the simulator at some fixed rates. We generate 4000 validator nodes and randomly distribute them into shards. In the current Bitcoin setting, the block size limit is 1 MB, and the average size of a transaction is 500 bytes, hence, each block contains approximately 2000 transactions. We evaluate the system performance with 4, 8, 12, and 16 shards, which are the number of shards that were used in previous studies [2], [3], [5].

4.2.1 Throughput

The experiment in this section illustrates how malicious transactions affect the system throughput. In order to find out the best throughput of the system, we gradually increase the transaction rate (i.e., the rate at which transactions are injected into the system) and observe the final throughput until the throughput stops increasing. At 16 shards, the best throughput is about 4000 tps, which is achieved when the transaction rate is about 5000 tps. For this experiment, we fix the rate at 5000 tps so that the system is always at its best throughput with respect to the number of shards.

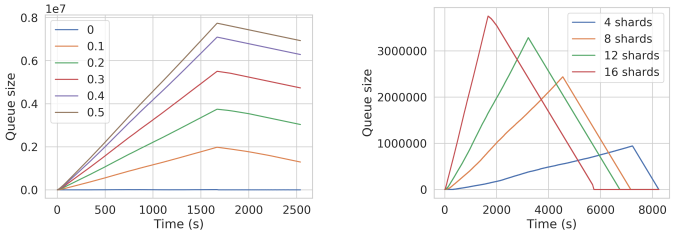
To perform the attack, the attacker runs Algorithm 1 to generate some portions of malicious transactions into shard 0. For example, if 10% of transactions are malicious, then at each second, 500 transactions will be put into shard 0, and the rest 4,500 txs are distributed into shards according to their hash value. The results are shown in the Fig. 5a.

At 0%, the system is not under attack, the system achieves its best throughput with respect to the number of shards. The horizontal dashed line illustrates the throughput of 1 shard, which is the lower bound of the system throughput. As can be seen, when we increase the number of malicious transactions, the system throughput rapidly decreases. This behavior can be explained as we have multiple cross-shard transactions that are associated with the attacked shard, their delays could produce a severe cascading effect that ends up hampering the performance of other shards. Thus, the throughput as a whole is diminished. Moreover, we can observe that higher numbers of shards are more vulnerable to the attack. With 16 shards, the performance reduces exponentially as we increase the portion of malicious transactions to 50%. Specifically, with only 20% malicious transactions, the throughput was reduced by more than half. This behavior confirms our prior preliminary analysis.

Another interesting observation is that at 50% malicious transactions, the throughput nearly reaches its lower bound, hence, the sharding system would not be any faster than using only 1 shard. At this time, the attacker has accomplished its goal, that is, making the attacked shard the bottleneck of the whole system. 50% malicious transactions translates to 2,500 malicious transactions that are sent to the system at each second, previous experiments and analysis in Table 1 show that a normal laptop could easily generate more than 100,000 malicious transactions per second.

4.2.2 Latency

We analyze the impact of the single-shard flooding attack on the system latency, which is the average amount of time needed to confirm a transaction. To avoid backlog when the system is not under attack, for each number of shards, the transaction rate is set to match the best system throughput. The results are shown in the Fig. 5b. In the same manner as the previous experiment, the attack effectively increases the latency of the system as a whole. We shall see in the next experiment that the attack creates a serious backlog in the mempool of the shards, thereby increasing the waiting time of transactions and eventually raising the average latency.



(a) Number of shards = 16 (b) 20% malicious transactions

Fig. 6: Impact on the attacked shard's queue size

This experiment also corroborates the experiment on system throughput as greater numbers of shards are also more vulnerable to the attack. With 16 shards, although it provides the fastest transaction processing when the system is not under attack, nevertheless, it becomes the slowest one even with only 10% of malicious transactions. Additionally, when the attacker generates 20% malicious transactions, the latency is increased by more than 10 times. Therefore, we can conclude that although adding more shards would help improve the system performance, under our attack, the system would become more vulnerable.

4.2.3 Queue/Mempool size

In the following experiments, we investigate the impact of this single-shard flooding attack on the queue (or mempool) size of shard 0, which is the shard that is under attack. This gives us insights into how malicious transactions cause a backlog in the shard. Firstly, we fix the number of shards and vary the portion of malicious transactions. Fig. 6a illustrates the queue size over time of shard 0 with a system of 16 shards where each line represents the portion of malicious transactions. When the system is not under attack, the queue size is stable with less than 15,000 transactions at any point in time. As we put in only 10% malicious transactions, the queue size reaches more than 2 million transactions.

Note that under our attack, the transactions are injected into shard 0 as a rate that is much higher than its throughput, thus, the queue will keep on increasing until all transactions have been injected. At this point, transactions are no longer added to the shard and the shard is still processing transactions from the queue, hence, the queue size decreases. This explains why the lines (i.e., queue size) go down towards the end of the simulation.

The result also demonstrates that the congestion gets worse as we increase the malicious transactions. Due to the extreme backlog, transactions have to wait in the mempool for a significant amount of time, thereby increasing their waiting time. This explains the negative impact of malicious transactions on system throughput and latency.

Next, we observe the queue size with different numbers of shards. Fig. 6b presents the impact of the attack with 20% malicious transactions at different numbers of shards. As can be seen, when we increase the number of shards, the backlog of transactions builds up much faster and greater due to the fact that we are having more cross-shard transactions. This result conforms to our previous claim that greater numbers of shards are more vulnerable to the attack.

4.2.4 Summary

The experiments presented in this section have shown that our attack effectively reduce the performance of the whole system by attacking only a single shard. By generating malicious transactions according to Algorithm 1, the attacker easily achieves its goal of limiting the system performance to the throughput of one shard. Our preliminary analysis in Section 3.2 shows that the attacker is totally capable of generating an excessive amount of malicious transactions at low cost, thereby demonstrating the practicality of the attack.

5 Countermeasure

As we have argued that using hash values to determine the output shards is susceptible to the single-shard flooding attack, we delegate the task of determining the output shards without using hash values to the validators. To achieve that task, we consider the validators running a deterministic transactions sharding algorithm. The program takes the form of $S_{out} = txsharding(tx, st)$ in which it ingests as inputs a blockchain state st and the transaction tx, and generates the output shard ID S_{out} of tx calculated at state st. Moreover, the algorithm $txsharding$ is made public. The requirement for $txsharding$ is that it is *deterministic* and has a *load-balancing* mechanism to balance the load among the shards. Finally, we assume that $txsharding$ does not use a transaction's hash as the basis for determining its output shard.

A simple approach to implement $txsharding$ is to devise an algorithm in which, for each input tx, S_{out} is determined by choosing the shard that is currently having the least number of transactions. In other words, $txsharding$ always distributes the transactions into shards evenly, thereby balancing the load among the shards. For an optimal $txsharding$, Nguyen et al. [5] proposes OptChain, a transactions placement algorithm that can both balance the load and minimize the number of cross-shard transactions. A technical overview of Optchain is given in Appendix B. With the load-balancing mechanism of $txsharding$, the attacker can no longer flood any one shard with a superfluous amount of transactions due to the fact that the transactions are always distributed evenly into shards.

However, the main challenge is how to run $txsharding$ so that we can prevent adversaries from manipulating its output. By the nature of blockchain, the validators are untrusted, a straw-man approach is to run $txsharding$ on-chain and let the validators reach consensus on the output of $txsharding$. Hence, the validators would have to reach consensus on every single transaction. Nonetheless, this alone dismisses the original idea of sharding, which is to improve blockchain by parallelizing the consensus work and storage, i.e., each validator only handles a disjoint subset of transactions. To avoid costly on-chain consensus on the output of $txshading$, we need to execute the algorithm off-chain while ensuring that the operation is tamper-proof in the presence of malicious validators. To tackle this challenge, in this work, we leverage the Trusted Execution Environment (TEE) to securely execute the transaction sharding algorithm on the validators.

TEE in a computer system is realized as a module that performs some verifiable executions in such a way that no other applications, even the OS, can interfere. Simply speaking, a TEE module is a trusted component within an untrusted system. In this work, we consider using TEE that supports issuing remote attestations proving the integrity of the software running inside the TEE module. Intel SGX [17] is one of the most commonly used implementations of TEE in which remote attestation is well-supported. However, TEE does not offer satisfactory availability guarantees as the hardware could be arbitrarily terminated by a malicious host or simply by losing power.

In the proposed countermeasure, validators are equipped with TEE modules that assist clients in determining the transactions' output shard with an attestation to prove the correctness of execution (most modern Intel CPUs from 2014 support Intel SGX). When a client issues a transaction to a validator, the validator will run its TEE module to get the output shard of that transaction, together with an *attestation* to prove the code's integrity and the correctness of the execution. The attestation is a digital signature generated by the TEE's private key (Section 5.2). The client can verify the computation using the attestation and then send the transaction with the attestation to the system in the same manner as issuing an ordinary transaction. With this concept, we can rest assured that an attacker cannot manipulate transactions to overwhelm a single shard. Additionally, we do not need the whole blockchain validators to reach consensus on a transaction's output shard, this computation is instead done off-chain by one or some small amount of validators. Its realization, however, encounters some challenges when using TEE in an untrusted network:

- A malicious validator can terminate the TEE at its discretion, which results in losing its state. The TEE module must be designed to tolerate such failure.
- Although the computation inside the TEE is trusted and verifiable via attestation, a malicious validator can deceive the TEE module by feeding it with fraudulent data.

To overcome these challenges, we aim to design a *stateless* TEE module where any persistent state is stored in the blockchain. To obtain the state from the blockchain, the TEE module acts as a blockchain client to query the block headers from the blockchain, thereby ensuring the correctness of the data (this is how we can exploit the immutability of blockchain to overcome pitfalls of TEE modules). With this design, even when some TEE modules are arbitrarily shut down, the security properties of the protocol are not affected.

5.1 System Overview and Security Goals

In this section, we present an overview of our system for the countermeasure and establish some security goals.

5.1.1 System overview

Our system considers two types of entities: clients and validators

- *Clients* are the end-users of the system who are responsible for generating transactions. The clients are not required to be equipped with a TEE-enabled platform. In fact, the clients in our system are extremely lightweight.

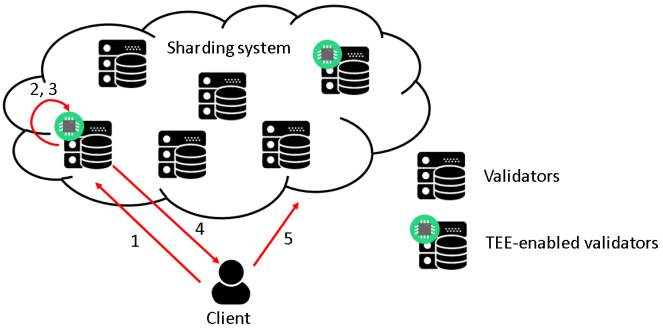


Fig. 7: System overview

- *Validators* in each shard maintain a distributed append-only ledger, i.e. a blockchain, of that shard with an intra-shard consensus protocol. Validators require a TEE-enabled platform to run the transaction sharding algorithm.

For simplicity, we assume that a client has a list of TEE-enabled validators and it can send requests to multiple validators to tolerate certain failures. Each TEE-enabled validator has $txsharding$ installed in its TEE module. We also assume that the TEE module in each of the validators constantly monitors the blockchain from each shard, so that it always has the latest state of the shards.

Denoting $ENC_k(m)$ as the encryption of message m under key k and $DEC_k(c)$ as the decryption of ciphertext c under key k, the steps for computing the output shard for a transaction is as follows (Fig. 7):

1) Client C sends the transaction tx to a TEE-enabled validators. C obtains the public key pk_{TEE} of a validator's TEE, computes $inp = ENC_{pk_{TEE}}(tx)$, and sends inp to the validator.
2) The validator loads inp into its TEE module and starts the execution of $txsharding$ in TEE.
3) The TEE decrypts the inp using its private key, and executes the $txsharding$ using tx and the current state st of the blockchain. Then, the output S_{out} is generated together with the state st upon which S_{out} is determined, and a signature σ_{TEE} proving the correct execution.
4) The validator then send $(S_{out}, st, \sigma_{TEE}, h_{tx})$ to C where h_{tx} is the hash of the transaction. C verifies σ_{TEE} before sending (σ_{TEE}, tx) to the blockchain network for final validation. If C sends a request to more than one validator, C would choose the S_{out} that reflects the latest state.
   Note that C could choose an outdated S_{out}, however, other entities can validate if a pair (S_{out}, st) is indeed the output of a TEE. The blockchain system can simply reject transactions whose S_{out} was computed based on an outdated st
5) Upon receiving (σ_{TEE}, tx), the validators again verify σ_{TEE} before proceeding with relaying and processing the transaction.

5.1.2 Adversarial model and Security goals

In the threat model in Section 3.1, the attacker only plays the role of a client, however, we stress that the countermeasure

must not violate the adversarial model of blockchain, which is working with malicious validators. Thus, in designing the countermeasure system, we extend the previous threat model as follows.

In the same manner as previous work on TEE-enabled blockchain [19], [20], we consider an adversary who controls the operating system and any other high-privilege software on the validators. Attackers may drop, interfere, or send arbitrary messages at any time during execution. We assume that the adversary cannot break the hardware security enforcement of TEE. The adversary cannot access processor-specific keys (e.g., attestation and sealing key) and it cannot access TEE's memory that is encrypted and integrity-protected by the CPU.

The adversary can also corrupt an arbitrary number of clients. Clients are lightweight, they only send requests to the validator the get the output shard of a transaction. They can verify the computation without TEE. We assume honest clients trust their platforms and software, but not that of others. We consider that the blockchain will perform prescribed computation correctly and is always available.

With respect to the adversarial model, we define the security notions of interest as follows:

1) Correct execution: the output of a TEE module must reflect the correct execution of $txsharding$ with respect to inputs tx and st, despite malicious host.
2) The system is secure against the aforementioned single-shard flooding attack.
3) Stateless TEE: the TEE module does not need to retain information regarding previous states or computations.

5.1.3 Blockchain sharding configuration

For ease of presentation, we assume a sharding system that resembles the OmniLedger blockchain [3]. Suppose the sharding system has n shards, for each $i \in \{1, 2, ..., n\}$, we denote BC_i and BH_i as the whole ledger and block headers of shard S_i, respectively. Furthermore, we assume that each shard S_i keeps track of its UTXO database, denoted by U_i. Given a shard S_i, each validator in S_i monitors the following database: BC_i, $BH_{j \neq i}$, and $U_{1,2,...,n}$.

As we use an Omniledger-like blockchain, each shard elects a leader who is responsible for accepting new transactions to the shard. For simplicity, we consider that the sharding system provides an API $validate(tx)$ that takes a transaction tx as the input and performs the transaction validation mechanism on tx. $validate(tx)$ returns true if tx is successfully committed to the blockchain, otherwise, it returns false.

Additionally, we consider the system uses a signature scheme $\Sigma(G, Sig, Vf)$ that is assumed to be EU-CMA secure (Existential Unforgeability under a Chosen Message Attack). ECDSA is a suitable signature scheme in practice [23]. Moreover, the hash function $\mathcal{H}(\cdot)$ used by the system is also assumed to be collision resistant: there exists no efficient algorithm that can find two inputs $a \neq b$ such that $\mathcal{H}(a) = \mathcal{H}(b)$.

Finally, we assume that each TEE generates a public/secret key pair and the public key is publicly available to all entities in the network. In practice, the public keys could be stored in a global identity blockchain.

Functionality $\mathcal{F}_{blockchain}$

Store $\mathbf{DB} = \{DB_i | i \in \{1, 2, ..., n\}\}$, each DB_i is an appended database indexed by h_{tx}.

- On **input** $read(id, h_{tx})$ from P_i: return $DB_{id}[h_{tx}]$ or $\perp$ if not exists.
- On **input** $write(id, tx)$ from P_i:
  1) If $validate(tx) = 0$ then **output** $reject()$.
  2) Otherwise, $DB_{id} = DB_{id} \parallel tx$ and **output** $accept()$

Fig. 8: Ideal blockchain $\mathcal{F}_{blockchain}$

5.2 Modeling Functionality of Blockchain and TEE

We specify the ideal blockchain $\mathcal{F}_{blockchain}$ as an appended decentralized database as in Fig. 8. $\mathcal{F}_{blockchain}$ stores $\mathbf{DB} = \{DB_i | i \in \{1, 2, ..., n\}\}$ which represents a set of blockchains that are held by shard 1, 2, ..., n, respectively. Each blockchain DB_i is indexed by the transactions' hash value h_{tx}. We assume that by writing to the blockchain of a shard, all validators of the shard reach consensus on that operation.

We specify the ideal TEE $\mathcal{F}_{TEE}$ that models a TEE module in Fig. 9, following the formal abstraction in [24]. On startup, $\mathcal{F}_{TEE}$ generates a public/secret key pair (pk_{TEE}, sk_{TEE}). Then, it publishes pk_{TEE} together with a remote attestation proof proving the integrity of the code in TEE, including the key generation code. With the public key, other entities in the network are able to verify messages signed by $\mathcal{F}_{TEE}$'s secret key. Since pk_{TEE} is bound to the TEE code, signatures under sk_{TEE} can be used to prove the integrity of the execution output. Therefore, they can be used as attestations for the correct execution of TEE.

In practice, TEE platforms like Intel SGX perform the remote attestation as follows. Suppose the execution on TEE results in an output and an attestation σ_{TEE}, as indicated in [20], the validator sends σ_{TEE} to the Intel Attestation Service (IAS) provided by Intel. Then IAS verifies σ_{TEE} and replies with $\pi = (b, \sigma_{TEE}, \sigma_{IAS})$, where b indicates whether σ_{TEE} is valid or not, and σ_{IAS} is a signature over b and σ_{TEE} by the IAS. Since π is basically a signature, it can be verified without using TEE or having to contact the IAS.

A TEE module is an isolated software container that is installed with a program that, in this work, is a transaction sharding algorithm. $\mathcal{F}_{TEE}$ abstracts a TEE module as a trusted third party for confidentiality, execution, and authenticity with respect to any entities that is a part of the system. $prog$ is a program that is installed to run in a TEE module; the input and output of $prog$ are denoted by inp and $outp$, respectively.

On initialization, the TEE module needs to download and monitor U_i for $i \in 1, 2, .., n$. These data are encrypted using the TEE's secret key and then stored in the host storage, which is also referred to as sealing. In this way, the TEE will make sure that its data on the secondary storage cannot be tampered with by a malicious host. To ensure that the TEE always uses the latest version of the sealed UTXO database, rollback-protection systems such as ROTE [25] can be used.

Functionality $\mathcal{F}_{TEE}$

- On **initialization**
  - Generate (pk_{TEE}, sk_{TEE})
- On **input** $install(prog)$ from P_k:
  - if $prog$ is not stored then $store(prog)$
- On **input** $resume(inp_c)$ from P_k:
  1) If $prog$ is not stored than return $\perp$.
  2) $outp = prog(inp_c)$
  3) $\sigma = \Sigma.Sig(sk_{TEE}, (prog, outp))$
  4) Return $(\sigma, outp)$ to P_k

Fig. 9: Ideal TEE $\mathcal{F}_{TEE}$

Program $prog$ run in the TEE module

Input: inp_c

1) Request the current state st from the sealed database
2) $tx \leftarrow DEC_{sk_{TEE}}(inp_c)$
3) $S_{out} = txsharding(tx, st)$
4) Return $(S_{out}, st, \mathcal{H}(tx))$

Fig. 10: Program $prog$ run in the TEE module

Fig. 10 defines the program $prog$ that is installed in the TEE module to be used in this work. As can be seen, the program decrypts the encrypted input inp_c using the secret key sk_{TEE}. $txsharding$ is implemented inside $prog$ to securely execute the transaction sharding algorithm. The program returns the output shard S_{out}, the state st upon which S_{out} was computed, and the hash of the transaction h_{tx}. This hash value is used to prevent malicious hosts from feeding the TEE module with fake transactions, which will be discussed in more detail in the next subsection.

Upon running $prog$ with the input inp_c, the TEE module obtains the signature σ_{TEE} over the output of $prog$ and the code of $prog$ using its private key. Finally, the TEE module returns to the host validator σ_{TEE}, and the output $outp$ from $prog$.

5.3 Formal Specification of the Protocol

Our proposed system supports two main APIs for the end-users: (1) $newtx(tx)$ handles the secure computation of a transaction tx, and (2) $read(id, h_{tx})$ returns the transaction that has the hash value h_{tx} from shard id.

The protocol for validators is formally defined in Fig. 11, which relies on $\mathcal{F}_{TEE}$ and $\mathcal{F}_{blockchain}$. The validator accepts two function calls from the clients: $request(tx_c)$ and $process(S_{out}, \sigma_{TEE}, tx)$. $request(tx_c)$ takes as input a transaction that is encrypted by the public key of TEE and sends tx_c to the TEE module. For simplicity, we assume that the validator is TEE-enabled, if not, the validator simply discards the $request(tx_c)$ function call. Since tx_c is encrypted by pk_{TEE}, a malicious host cannot tamper with the transaction. The validator waits until the TEE returns an output and relays that output to the function's caller.

Note that as the output of the TEE includes the transaction's hash h_{tx}, the client can check that the TEE indeed

Protocol for validators

- On **input** $request(tx_c)$ from C_i:
  1) Send $resume(tx_c)$ to $\mathcal{F}_{TEE}$
  2) Wait to receive $(\sigma_{TEE}, (S_{out}, st, h_{tx}))$ from $\mathcal{F}_{TEE}$
  3) Return $(st, S_{out}, \sigma_{TEE}, h_{tx})$
- On **input** $process(S_{out}, \sigma_{TEE}, tx)$ from C_i:
  1) Assert σ_{TEE} is valid, if failed then return $\perp$
  2) Send $write(S_{out}, tx)$ to $\mathcal{F}_{blockchain}$
  3) Wait to receive output from $\mathcal{F}_{blockchain}$
  4) Return the received output to C_i

Fig. 11: Protocol for validators

Protocol for clients

- On **input** $newtx(tx)$ from environment $\mathcal{Z}$:
  1) $tx_c \leftarrow ENC_{pk_{TEE}}(tx)$
  2) Send $request(tx_c)$ to validator P_k
  3) Receive $(st, S_{out}, \sigma_{TEE}, h_{tx})$
  4) Assert $\mathcal{H}(tx) = h_{tx}$
  5) Assert σ_{TEE} using $(st, S_{out}, txsharding, \mathcal{H}(tx))$, if fail then return $\perp$
  6) Send $process(S_{out}, \sigma_{TEE}, tx)$ to S_{out}
  7) Wait to receive $accept()$ or $\perp$ from P_k
  8) Forward the received data to $\mathcal{Z}$
- On **input** $read(id, idx)$ from environment $\mathcal{Z}$:
  1) Send $read(idx)$ to $\mathcal{F}_{blockchain}[id]$ and relay output to $\mathcal{Z}$

Fig. 12: Protocol for clients

processed the correct transaction tx originated from the client. This is possible because of the end-to-end encryption of tx between the client and the TEE. Furthermore, since σ_{TEE} protects the integrity of S_{out}, the client can verify that S_{out} was not modified by a malicious validator.

The function $process(S_{out}, \sigma_{TEE}, tx)$ receives as input the transaction tx, σ_{TEE}, and output shard S_{out} of tx. The validator also verifies σ_{TEE} before making a call to $\mathcal{F}_{blockchain}$ to start the transaction validation for tx.

Fig. 12 illustrates the protocol for the clients. To determine the output shard of a transaction tx, a client invokes the API $newtx(tx)$. First, to ensure the integrity of tx, the client encrypts tx using the pk_{TEE} and sends tx_c to a TEE-enabled validator P_k. Upon receiving $(st, S_{out}, \sigma_{TEE}, h_{tx})$ from P_k, the client checks if the hash of tx is equal to h_{tx}. This prevents a malicious validator from feeding a fake transaction to the TEE module to manipulate S_{out}. The client also verifies if the attestation σ_{TEE} is correct. Afterward, the client sends the transaction together with S_{out} and the attestation to the validators of the transaction's input and output shards for final validation. $newtx(tx)$ finally outputs any data received from the validators. The API $read(id, h_{tx})$ can be called when the client wants to obtain the transaction information from the blockchain. The function also returns any data received from $\mathcal{F}_{blockchain}$.

6 Security Analysis and Performance Evaluation

This section presents a detailed security analysis of the proposed countermeasure under the UC-model and evaluates the performance of the proof-of-concept implementation.

6.1 Security Analysis

We first prove that the proposed protocol (1) only requires a stateless TEE and (2) is secure against the single-shard flooding attack, and then prove the correct execution security. By design, the $txsharding$ installed in the TEE depends solely on the transaction tx, and the state st of the blockchain for computation. As tx is the input and st can be obtained by querying from the UTXO database, the TEE does not need to keep any previous states and computation, thus, it is stateless.

When the system makes decision on the output shard of a transaction, it relies on the $txsharding$ program which is assumed to not base its calculation on transactions' hash value. Therefore, as $txsharding$ also balances the load among the shards, no attackers can manipulate transactions to overwhelm a single shard, hence, the countermeasure is secure against the single-shard flooding attack.

The correct execution security of our system is proven in the Universal Composability (UC) framework [26]. We refer the readers to Appendix A for our proof.

6.2 Performance Evaluation

This section presents our proof-of-concept implementation as well as some experiments to evaluate its performance. As our countermeasure is immune to the single-shard flooding attack, our goal is to evaluate the overhead of integrating this solution into sharding. We implement the proof-of-concept using Intel SGX which is available on most modern Intel CPUs. With SGX, each implementation of the TEE module is referred to as an *enclave*. The proof-of-concept was developed on Linux machines in which we use the Linux Intel SGX SDK 2.1 for development. We implement and test the protocol for validators using a machine equipped with an Intel Core i7-6700, 16GB RAM, and an SSD drive.

As we want to demonstrate the practicality of the countermeasure, the focus of this evaluation is three-fold: processing time, communication cost, and storage. The processing time includes the time needed for the enclave to monitor the block headers as well as to determine the output shard of a transaction requested by a client. The communication cost represents the network overhead incurred by the interaction between clients and validators to determine the output shards. Additionally, we measure the amount of storage needed when running the enclave.

In our proof-of-concept implementation, we use OptChain [5] as $txsharding$. As OptChain determines the output shard based on the transaction's inputs, when obtaining the state from the UTXO database, we only need to load those transaction's inputs from the database. Our proof-of-concept uses Bitcoin as the blockchain platform.

Processing time. We calculate the processing time for determining the output shard by invoking the enclave with 10 million Bitcoin transactions (encrypted with the TEE public key) and measure the time needed to receive output from the enclave. This latency includes (1) decrypting the transaction, (2) obtaining the latest state from the UTXO database in the host storage and (3) running $txsharding$. We observe that the highest latency recorded is only about 214 ms and it also does not vary much when running with different transactions. Considering that the average latency of processing a transaction in sharding is about 10 seconds [5], our countermeasure only imposes an additional 0.2 seconds for determining the output shard.

For a detailed observation, we measure the latency separately for each stage. The running time of $txsharding$ is negligible as the highest running time recorded is about 0.13 ms when processing a 10-input transaction at 16 shards. Decrypting the transaction is about 0.579 ms when using 2048-bit RSA, and a query to the UTXO database to fetch the current state takes about 213.2 ms. Hence, fetching the state from the UTXO database dominates the running time due to the fact that the database is stored in the host storage.

Another processing time that we consider is the time needed for updating the UTXO database when a new block is added to the blockchain. With an average number of 2000 transactions per block, each update takes about 65.7 seconds. But this latency does not affect the performance of the system since we can set up two different enclaves running in parallel, one for running $txsharding$, and one for monitoring the UTXO database. Therefore, the time needed to run $txsharding$ is independent of updating the database.

Communication cost. Our countermeasure imposes some communication overhead over hash-based transaction sharding since the client has to communicate with the validator to determine the output shard. Specifically, the overhead includes sending the encrypted transaction to the validator and receiving a response. The response comprises the state (represented as the block number), output shard ID, attestation, and a hash value of the transaction. The communication overhead sums up to about 601 bytes needed for the client to get the output shard of a transaction. If we consider a communication bandwidth of 10 Mbps, the transmission time would take less than half a millisecond.

Storage. According to the proposed system for the countermeasure, the enclave needs to store the UTXO database in the host storage, which is essentially the storage of the validator. As of Feb 2020, the size of Bitcoin's UTXO set is about 3.67 GB, which means that an additional 3.67 GB is needed in the validator's storage. However, considering that the validator's storage is large enough to store the whole ledger, which is about 263 GB as of Feb 2020, the extra data trivially accounts for 1.4%.

Summary. By evaluating a proof-of-concept of our countermeasure, the result in this section shows that our system imposes negligible overhead compared to the hash-based transaction sharding, which is susceptible to the single-shard flooding attack. Particularly, by incurring insignificant processing time, communication cost, and storage, our proposed countermeasure demonstrates the practicality as it can be integrated into existing sharding solutions without affecting the system performance.

7 Conclusion

In this paper, we have identified a new attack in existing sharding solutions. Due to the use of hash-based transaction

sharding, an attacker can manipulate the hash value of a transaction to conduct the single-shard flooding attack, which is essentially a DoS attack that can overwhelm one shard with an excessive amount of transactions. We have thoroughly investigated the attack with multiple analyses and experiments to illustrate its damage and practicality. Most importantly, our work has shown that by overwhelming a single shard, the attack creates a cascading effect that reduces the performance of the whole system.

We have also proposed a countermeasure based on TEE that efficiently eliminates the single-shard flooding attack. The security properties of the countermeasure have been proven in the UC framework. Finally, with a proof-of-concept implementation, we have demonstrated that our countermeasure imposes negligible overhead and can be integrated into existing sharding solutions.

REFERENCES

[1] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 17–30.

[2] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 931–948.

[3] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[4] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 95–112.

[5] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: optimal transactions placement for scalable blockchain sharding," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 525–535.

[6] C. Huang, Z. Wang, H. Chen, Q. Hu, Q. Zhang, W. Wang, and X. Guan, "Repchain: A reputation-based secure, fast, and high incentive blockchain system via sharding," *IEEE Internet of Things Journal*, vol. 8, no. 6, pp. 4291–4304, 2020.

[7] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 90–105.

[8] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered sharding blockchain system," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[9] A. Manuskin, M. Mirkin, and I. Eyal, "Ostraka: Secure blockchain scaling by node sharding," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2020, pp. 397–406.

[10] M. Saad, L. Njilla, C. Kamhoua, J. Kim, D. Nyang, and A. Mohaisen, "Mempool optimization for defending against ddos attacks in pow-based blockchain systems," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2019, pp. 285–292.

[11] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, "Stressing out: Bitcoin "stress testing"," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 3–18.

[12] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[13] Z. Tan, A. Jamdagni, X. He, P. Nanda, and R. P. Liu, "A system for denial-of-service attack detection based on multivariate correlation analysis," *IEEE transactions on parallel and distributed systems*, vol. 25, no. 2, pp. 447–456, 2013.

[14] J. Yu, H. Lee, M.-S. Kim, and D. Park, "Traffic flooding attack detection with snmp mib using svm," *Computer Communications*, vol. 31, no. 17, pp. 4212–4219, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366408005094

[15] Z. A. Biron, S. Dey, and P. Pisu, "Real-time detection and estimation of denial of service attack in connected vehicle systems," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 12, pp. 3893–3902, 2018.

[16] "Report: Bitcoin (btc) mempool shows backlogged transactions, increased fees if so?" June 2018. [Online]. Available: https://bitcoinexchangeguide.com/report-bitcoin-btc-mempool-shows-backlogged-transactions-increased-fees-if-so/

[17] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," vol. 13, p. 7, 2013.

[18] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "Bite: Bitcoin lightweight client privacy using trusted execution," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 783–800.

[19] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, "Teechain: a secure payment network with asynchronous blockchain access," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 63–79.

[20] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.

[21] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi, "Fastkitten: practical smart contracts on bitcoin," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 801–818.

[22] Y. Aoki, K. Otsuki, T. Kaneko, R. Banno, and K. Shudo, "Simblock: A blockchain network simulator," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2019, pp. 325–329.

[23] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[24] R. Pass, E. Shi, and F. Tramer, "Formal abstractions for attested execution secure processors," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 260–289.

[25] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "Rote: Rollback protection for trusted execution," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1289–1306.

[26] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 2001 IEEE International Conference on Cluster Computing*. IEEE, 2001, pp. 136–145.

$\mathcal{F}_{cm}$: Ideal functionality of the countermeasure

- On **Initialization**:
  1) $DB_i = \emptyset, \forall i \in [N]$
- On **input** $newtx(tx)$ from any party P_i:
  1) relay input to $\mathcal{A}$
  2) Request current state st from the UTXO database
  3) $S_{out} = txsharding(tx, st)$
  4) $DB_{S_{out}} = DB_{S_{out}}||tx$
  5) Send delayed $accept()$ to P_i
- On **input** $read(id, h_{tx})$ from any party P_i:
  1) If $DB_{id}[h_{tx}]$ is not available then return $\perp$
  2) Otherwise, return $DB_{id}[h_{tx}]$

Fig. 13: Ideal functionality of the countermeasure

APPENDIX A
PROOF OF CORRECT EXECUTION IN THE UC FRAMEWORK

In the UC framework, a real world involves parties running the proposed protocol, namely Π_{cm}. On the other hand, an ideal world consists of parties that interact with an ideal functionality $\mathcal{F}_{cm}$, a trusted third party that implements the APIs of the proposed protocol, i.e., $newtx(\cdot)$ and $read(\cdot)$. Fig. 13 shows the definition of $\mathcal{F}_{cm}$. Any adversary $\mathcal{A}$ in the real world is introduced in the ideal world by a simulator $\mathcal{S}$ with an adversary model defined in Section 5.1.2.

To prove that the proposed protocol Π_{cm} achieves correct execution security, we show that: (1) $\mathcal{F}_{cm}$ achieves the correct execution security in the ideal world; and (2) the real and ideal worlds are indistinguishable to an external environment $\mathcal{Z}$. This implies that any attack violating security goals in the real world is translatable to a corresponding attack in the ideal one. This proves that the real world protocol also achieves the correct execution security.

In $\mathcal{F}_{cm}$, whenever a party triggers $newtx(tx)$ with a transaction tx, the execution of $txsharding$ is performed internally by the ideal functionality $\mathcal{F}_{cm}$ based on tx and the current state st to determine S_{out}. Since $\mathcal{F}_{cm}$ is trusted under UC, $txsharding$ is guaranteed to be correctly executed. Furthermore, $\mathcal{F}_{cm}$ also validates the transaction in the blockchain and returns only $accept()$ or $\perp$ to the party, an adversary does not have control over the output shard S_{out}, hence, the adversary cannot tamper with S_{out}. Therefore, $\mathcal{F}_{cm}$ achieves correct execution security.

Let $\mathcal{A}$ be an adversary against the proposed protocol. Per Canetti [26], we say that Π_{cm} UC-realizes $\mathcal{F}_{cm}$ if there exists a simulator $\mathcal{S}$, such that any environment $\mathcal{Z}$ cannot distinguish between interacting with the adversary $\mathcal{A}$ and Π_{cm} or with the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}_{cm}$. By that definition, we prove the following theorem:

Theorem 1. *The protocol Π_{cm} in the $(\mathcal{F}_{TEE}, \mathcal{F}_{blockchain})$ hybrid model UC-realizes the ideal functionality $\mathcal{F}_{cm}$.*

Proof. We prove the indistinguishability between the real and ideal worlds through a series of *hybrid steps* as commonly done in previous work [19], [20]. These hybrid steps start at H_0 - the real-world execution of Π_{cm} in the $(\mathcal{F}_{TEE}, \mathcal{F}_{blockchain})$ hybrid model, and finally becomes the execution in the ideal world. The indistinguishability is proven in each step.

Hybrid H_0 is the real-world execution where parties run Π_{cm} in the $(\mathcal{F}_{TEE}, \mathcal{F}_{blockchain})$ hybrid model.

Hybrid H_1 behaves in the same manner as H_0, except that $\mathcal{S}$ emulates $\mathcal{F}_{TEE}$ and $\mathcal{F}_{blockchain}$. First, $\mathcal{S}$ generates a key pair (pk_{TEE}, sk_{TEE}) and publishes pk_{TEE}. Whenever $\mathcal{A}$ interacts with $\mathcal{F}_{TEE}$, $\mathcal{S}$ records messages sent by $\mathcal{A}$ and emulates $\mathcal{F}_{TEE}$'s behavior. Likewise, $\mathcal{S}$ emulates $\mathcal{F}_{blockchain}$ by storing $\mathbf{DB}$ internally. As the view of $\mathcal{A}$ in H_1 is identically simulated in H_0, $\mathcal{Z}$ cannot distinguish between H_1 and the execution H_0.

Hybrid H_2 proceeds as H_1. However, every time $\mathcal{A}$ communicates with $\mathcal{F}_{blockchain}$, $\mathcal{S}$ identically emulates $\mathcal{F}_{blockchain}$'s behavior for $\mathcal{A}$. As the view of $\mathcal{A}$ in H_2 are simulated when interacting with the ledger, then environment $\mathcal{Z}$ cannot distinguish between H_2 and H_1.

Hybrid H_3 modifies H_2 as follows. When $\mathcal{A}$ triggers $\mathcal{F}_{TEE}$ with a message $install(prog)$, $\mathcal{S}$ records a tuple $(\sigma_{TEE}, outp)$ for all subsequent $resume(\cdot)$ calls, where $outp$ is the output of $prog$ and σ_{TEE} is an attestation under sk_{TEE} over $outp$ and $prog$. $\mathcal{S}$ keeps a set of all such tuples. Whenever $\mathcal{A}$ sends a tuple $(\sigma_{TEE}, outp)$ that has not been recorded by $\mathcal{S}$ to $\mathcal{F}_{blockchain}$ or an honest party, $\mathcal{S}$ simply stops the execution.

We can prove that $\mathcal{Z}$ cannot distinguish between H_3 and H_2 as follows. In H_2, if $\mathcal{A}$ sends forged attestations/signatures to $\mathcal{F}_{blockchain}$ or an honest party, signature verification by $\mathcal{F}_{blockchain}$ or the honest party will fail with negligible probability (as we assume the signature scheme Σ is EU-CMA secure). If $\mathcal{Z}$ can distinguish H_2 from H_3, we can construct an adversary using $\mathcal{Z}$ and $\mathcal{A}$ to win the game of signature forgery.

Hybrid H_4 proceeds as H_3 with one modification: $\mathcal{S}$ emulates the new transaction processing. Specifically, honest parties send $newtx$ to $\mathcal{F}_{cm}$. $\mathcal{S}$ emulates messages from $\mathcal{F}_{TEE}$ and $\mathcal{F}_{blockchain}$ as in H_3, i.e., recording tuples $(\sigma_{TEE}, outp)$. If the party is corrupted, $\mathcal{S}$ sends $newtx(tx)$ to $\mathcal{F}_{cm}$ as P_i. It can be seen that the view of $\mathcal{A}$ is the same as in H_2, as $\mathcal{S}$ can identically emulate $\mathcal{F}_{TEE}$ and $\mathcal{F}_{blockchain}$.

It can be seen that H_4 is identical to the ideal protocol. In H_4, while $\mathcal{S}$ interacts with $\mathcal{F}_{cm}$, it emulates $\mathcal{A}$'s view of the real-world. Now, $\mathcal{S}$ only needs to output to $\mathcal{Z}$ what $\mathcal{A}$ outputs in the real-world. Thus, there exists no environment $\mathcal{Z}$ that can distinguish between interaction between $\mathcal{A}$ and Π_{cm}, from interaction between $\mathcal{S}$ and $\mathcal{F}_{cm}$. □

APPENDIX B
TECHNICAL OVERVIEW OF OPTCHAIN

Nguyen et al. [5] proposes a transaction placement algorithm to improve the throughput of sharding blockchain, called OptChain. OptChain introduces a new sharding paradigm, in which cross-shard transaction are minimized while maintaining the load balancing among shards, resulting in almost twice faster confirmation time and throughput. Specifically, they form an optimization problem in which the algorithm determines the best shard to submit a transaction in order to minimize cross-shard transactions while guaranteeing the temporal balance, thereby shortening the confir-

mation time and boosting the overall system throughput. To achieve that, the algorithm optimizes the following goals:

1) Minimizing cross-shard transactions: Reduce the number of cross-shard transactions by grouping related transactions into a same shard.
2) Maintaining temporal balancing: To distribute load evenly among shards to increase parallelism and reduce queuing time.

On the one hand, by treating transactions as a stream of nodes in online directed acyclic graph (DAG), they propose a lightweight and on-the-fly PageRank score calculation to quantitatively measure transactions' relationship to each shard. This is referred to as the *Transaction-to-Shard* (T2S) score. The T2S score between a transaction u and a shard S_i measures the probability that a random walk from the node u ends up at some node in S_i, i.e., how likely the transaction should be placed into the shard without causing further cross-shard transactions.

On the other hand, they provide an estimation on the expected latency if putting a transaction into a given shard, which is called the *Latency-to-Shard* (L2S) score. The L2S score estimates the processing delay when placing the transaction into a shard.

The decision making of which shard to place a transaction into is guaranteed to balance both the T2S and L2S scores. With this mechanism, OptChain has been shown to outperform Omniledger, which uses transactions' hash values to determine their output shard. Furthermore, OptChain's throughput can be scaled up linearly with the number of shards while still keeping almost the same average transaction confirmation delay.