Code is the (F)Law: Demystifying and Mitigating Blockchain Inconsistency Attacks Caused by Software Bugs

Guorui Yu[¶*], Shibin Zhao[†*], Chao Zhang[‡⊠], Zhiniang Peng[§], Yuandong Ni[‡] and Xinhui Han[¶⊠]
[¶]Peking University, yuguorui@pku.edu.cn, hanxinhui@pku.edu.cn
[†]State Key Laboratory of Mathematical Engineering and Advanced Computing, zsbpro@163.com
[‡]Tsinghua University, chaoz@tsinghua.edu.cn, nyd17@tsinghua.org.cn
[‡]Tsinghua University-QI-ANXIN Group JCNS
[§]Sangfor Technologies Inc., jiushigujiu@gmail.com
[§]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488749

Abstract—**Blockchains promise to provide a tamper-proof medium for transactions, and thus enable many applications including cryptocurrency. As a system built on consensus, the correctness of a blockchain heavily relies on the consistency of states between its nodes. But consensus protocols of blockchains only guarantee the consistency in the transaction sequence rather than nodes' internal states. Instead, nodes must replay and execute all transactions to maintain their local states independently. When executing transactions, any different execution result could cause a node out-of-sync and thus gets isolated from other nodes.**

After systematically modeling the transaction execution process in blockchains, we present a new attack INCITE, which can lead different nodes to different states. Specifically, attackers could invoke an ambiguous transaction of a vulnerable smart contract, utilize software bugs in smart contracts to lead nodes that execute this transaction into different states. Unlike attacks that bring short-term inconsistencies, such as fork attacks, INCITE can cause nodes in the blockchain to fall into a long-term inconsistent state, which further leads to great damages to the chain (e.g., double-spending attacks and expelling mining power). We have discovered 7 0day vulnerabilities in 5 popular blockchains which can enable this attack. We also proposed a defense solution to mitigate this threat. Experiments showed that it is effective and lightweight.

I. Introduction

Emerging blockchain technology has spawned a large number of distributed applications, in which the smart contract plays a key role. The main reason for the success of smart contract is its baked-in decentralization. Instead of relying on a single instance to regulate assets, blockchain systems use the "state machine replication"[1] of nodes that use consensus protocols to reach consistency on a public distributed ledger of transactions. Almost all blockchain projects claim that they can resist attacks and prevent the blockchain or smart contracts from being placed in an inconsistent or incorrect state, and guarantee the underlying consensus protocol does not get compromised by the attacker. But this claim is built based on a crucial assumption: *different nodes process transactions correctly and consistently*, which requires that nodes written in different languages and running on various platforms must fully agree on the results of smart contract execution.

Although in recent years we have seen a lot of work [2], [3], [4] about vulnerability analysis of smart contracts, few attentions have been paid to the transaction execution layer that is used to execute smart contracts and apply changes to local state based on transactions. The transaction execution layer can be viewed as a state machine that receives a transaction as input and produces a new state as output. Since the state is not explicitly encoded in transactions, nodes must replay all transactions orderly to maintain the local state independently, including transaction invocations and smart contract creations. To ensure the consistency of the state between nodes, the result of smart contract execution must be deterministic and determined only by the input transaction and the previous state, and should not have different results even in nodes running code in different languages or architectures. However, input contracts deployed in the blockchain have diversified sources and many of them are not trusted and could introduce security risks to nodes.

INCITE Attacks. The consensus protocol can only guarantee the integrity of the block sequence, rather than the consistent execution and state consistency. In this paper, we present *INCITE attacks*, where the attacker constructs an ambiguous transaction and broadcast it to the network, causes different execution results of smart contracts in different nodes, thereby misleading the victim nodes to a maliciously constructed state. The attacker can then further exploit the corrupted state, such as obstructing victim nodes to form consensuses with the rest of nodes in the network, forcing victim nodes to waste computing power on invalid views of the blockchain, splitting the network into multiple pieces or a building block of another attack (expelling mining power, double-spending attack, etc., see Section IV-C). Moreover, the ambiguous transaction can be constructed by utilizing low-threaten "trivial" vulnerabilities (e.g., out-of-buffer read, uninitialized variables) or some minor implementation differences in different platforms. To evaluate the impact of INCITE attacks, we audited several existing well-known blockchain projects (EOS [5], NEO [6], and ONT [7]) and discovered 7 zero-day vulnerabilities which are feasible to achieve the INCITE attacks (Section IV-A).

Two reasons make INCITE attacks non-negligible: (1) inevitable software vulnerabilities, (2) distinctions among different client implementations. An uninitialized variable generally does not directly cause severe problems in a typical

[*] The two lead authors contributed equally to this work.
Email: hanxinhui@pku.edu.cn, chaoz@tsinghua.edu.cn.

978-1-6654-0325-2/21/$31.00 ©2021 IEEE

application, but different nodes may produce different execution results if a smart contract can access this uninitialized variable. Similarly, differences in platforms and programming languages can lead to inconsistent execution of nodes. Such vulnerabilities have been triggered accidentally before. Indeed, the clients of Ethereum (Geth and Parity) were once unable to accept each other's transactions and blocks and split the network because of the difference in implementation details [8]. Compared with the Eclipse attack [9], which focuses on attacking the blockchain's P2P network, our INCITE attacks focus on *the transaction execution stage* of the blockchain.

The INCITE attacks are difficult to detect without explicit protective measures because the smallest unit of information synchronization between nodes is a transaction rather than a temporal memory state of nodes, and the memory state is out of the protection of consensus protocol. Therefore, the victim node cannot immediately find the out-of-synchronization of the local state after processing the problematic transaction, but can only verify the state indirectly through the following transactions caused by the error state. Besides, not all operations to nodes will be forwarded to the network, such as some "read-only" operations or rolled-back transactions, but they may cause changes in the state of the node.

Countermeasures. To prevent attackers from exploiting corrupted states, we need a mechanism to verify the transaction execution results and the modification to the local state, and this seems to be easily solved by adding verification filed to the chain. However, there are several challenges to overcome. First, how should we generate the verification filed effectively to maintain performance among block increasing? Second, where should the field be, and how can we ensure that this field is unable to bypass? Third, what should nodes do when they find that the state is invalid? Is it safe to only discard illegal blocks? Thus, our next contribution is that we propose a general lightweight countermeasure to mitigate INCITE attacks (Section VI). The experiment result turns out that our mitigation can resist the INCITE attack effectively. The CPU performance overhead is 3.8% and the additional storage required in block is negligible.

To summarize, we make the following contributions.

1) We proposed and systematically studied a new kind of attacks, the INCITE attack, which could drive the blockchain into long-term inconsistent states, and modeled the attacks and analyzed the root causes of these attacks.
2) We discovered a new way to exploit the blockchain system to achieve double-spending with vulnerabilities that were previously considered unusable.
3) We discovered 7 zero-day vulnerabilities in 5 popular blockchain projects and proved the feasibility of the INCITE attack.
4) We proposed a defense solution to mitigate the INCITE attack, and evaluated its performance and security.

II. Background

This section introduces the basic data structures in the blockchain, such as Blocks, Transactions, etc. These different data structures are reflected in different levels of abstraction in the blockchain system. We will also introduce the general architecture (Section II-B) of the blockchain system to introduce INCITE attacks.

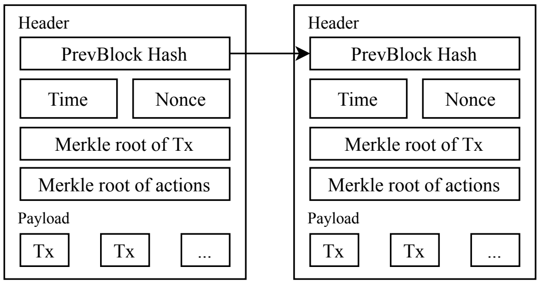

Fig. 1. Data structure of a Blockchain. Different blockchains usually follow a similar data structure of block, where some extra fields (eg., nonce, witness) need be added for adoption of the PoW or PoS.

A. Blockchain data structure

As shown in Fig.1, a typical *block* in blockchain is composed of a *header* and a *payload*. The header includes critical information, such as the hash of the previous block, the Merkle root of transactions, and the timestamp. The transactions corresponding to the Merkle roots are organized in the payload.

Transaction. Transactions are the underlying carrier of blockchain data. The account balance is not explicitly maintained by blocks or a single transaction, but by calculating from the historical transactions. A transaction primarily consists of 4 parts: *inputs*, *outputs*, *data* and *signature*.

Inputs are references to previous transactions that transfer digital assets to the current account. But it is worth noting that each transaction that has not spent can only be cited once. We call the collection of all transactions that are unspent as Unspent Transaction Output (UTXO). When an attacker attempts a double-spending attack, the essence is to refer to the same UTXO in multiple transactions.

B. Blockchain system architecture

There are some existing studies on attacks against blockchains, e.g., the 51% attack[10], the eclipse attack[11]. However, they mainly focused on the consensus layer or the P2P layer, and few have paid attentions to the transaction execution layer.

Network layer. Most blockchain systems usually choose the P2P network as the network layer with considering the distributed design of the blockchain system. Although communication between nodes may be encrypted, an attacker can still introduce malicious nodes into the network to carry out attacks. If a node is induced to connect to a maliciously connected node, it may suffer from an eclipse attack and eventually lead to an obsolete view of the blockchain or a double-spending attack. Similar attacks on the network layer include DNS hijacking and BGP hijacking [12].

Consensus layer. The consensus layer is the cornerstone of blockchain security. A mature and reliable blockchain system should not only address the problem of failing nodes, but also

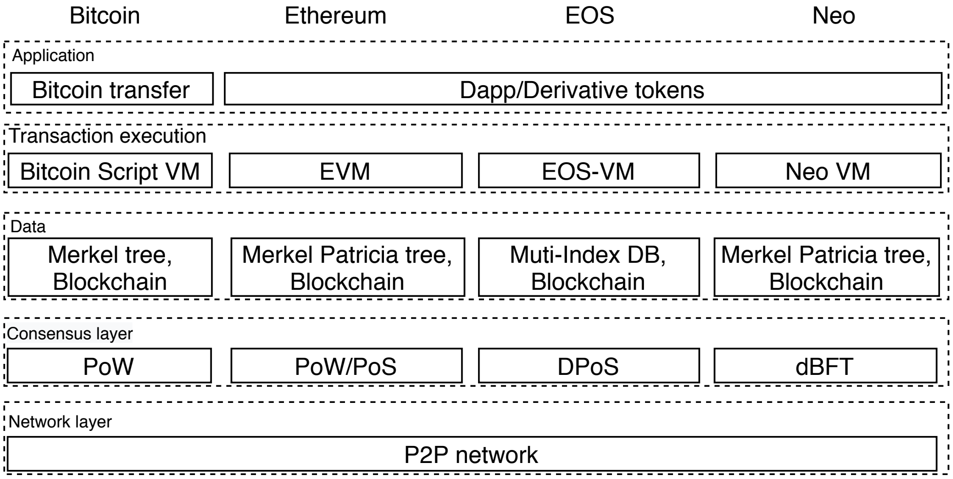

Fig. 2. Blockchain system architecture.

address the problem of untrusted nodes, i.e., Byzantine fault-tolerant [13]. The open blockchain system also needs to deal with Sybil attacks [14] caused by the free access of nodes.

Data layer. The data layer works above the consensus protocol, and the nodes need a specific data structure as a carrier for synchronization. In the blockchain system, the primary data structure of consensus protocol synchronization is the block. More specifically, each block in the blockchain is divided into two parts: the block header and the block body. The block header contains only some critical data (such as the Merkle root of all transactions) to ensure the immutability of recorded transactions. And the actual transaction data is stored in the block body. Bitcoin, Ethereum, and most other currencies use transaction-based data models.

It is worth noting that the consensus layer only guarantees the consistency of blocks or transactions, but does not guarantee the consistency of the state generated by the transaction execution, which lays a security hazard for the blockchain.

Transaction execution layer. The transaction execution layer can be thought of as a state transition system that maintains all currency ownership, persistent data of smart contracts, and so on.

The virtual machine, such as EVM, EOS-VM or Neo VM, is the key component to ensure the consistency of the blockchain. Most virtual machine follows the stack machine architecture, but with some customizations towards the peculiarities of blockchain. Once the consistency of the transaction execution layer is compromised, it will affect the consistency of the network, but there is no previous work related to this field.

III. The transaction workflow

The transaction workflow plays an important role in the INCITE attack. We analyzed some blockchain projects (EOS, NEO, and ONT) and summarized their models as follows.

A. Proposal phase

After a user constructs and signs a transaction, the proposed transaction is forwarded to the P2P network, then the nodes in the network will try to include it to the current block to gain more reward. Once the transaction is adopted, the endorsing nodes execute the transaction locally, calculate the Merkle root of transactions and *actions* (Section II-A), and package the executing result in the block. Due to the possible differences between nodes and *transaction-agnostic* of the consensus scheme, blocks containing the same transaction but different execution results may spread on the P2P network simultaneously.

TABLE I
Vulnerabilities used to construct INCITE attacks

CVE ID	Projects	Descriptions
CVE-2017-14457	Ethereum	Out-of-bound read
CVE-2018-20421	Ethereum	Computing result related to the memory size
CVE-2018-20690	EOS	Out-of-bound read
CVE-2018-20689	EOS	Memory corruption
CVE-2018-20696	EOS	Uninitialized memory
CVE-2018-20692	EOS	Inconsistent version implementation
CVE-2018-20693	NEO	Leaky abstraction of APIs
CVE-2018-20746	NEO	Different implementation in languages
CVE-2018-20694	ONT	Different implementation of the cryptographic algorithm

B. Validation phase

After receiving blocks, all nodes in the blockchain individually validate the transactions within the received blocks before commit changes to the ledger. The node first performs sanity checks, such as some fields like block size or height. After this, the node will check the block's existence in the ledger according to the block hash, and if it already exists, the current block will be skipped. If not, the check of consensus rules will be performed.

If the block does not pass the integrity or consensus rule check during the verification process, it will only be skipped or discarded without further processing.

C. Commit phase

After the block passes the verification phase, it will enter this phase to commit the modification to the ledger. There are some subtle differences between different types of consensus protocols.

Chain-based consensus. For a chain-based consensus protocol, there may be multiple forks in the network at the same time. When a node chooses a branch, the candidate needs to be valid and with the highest difficulty. Ambiguity transactions will cause forks in chain-based consensus protocols, and these forks do not agree with each other's legitimacy, regardless of their total difficulty.

BFT-based consensus. BFT-based consensus protocols do not develop forks during the consensus process, but it may still be exploitable by ambiguous transactions. The reason is that some blockchains will not record the results of transaction execution. When an ambiguous transaction is encountered, an implicit fork occurs between the nodes. In other words, although nodes share the same transaction sequence, the state between nodes is not synchronized. We will cover more details in Section IV-C.

IV. The INCITE attack

For INCITE attacks, the most critical is the construction of ambiguous transactions which will lead to a network split implicitly or explicitly. We have found 7 zero-day vulnerabilities which any of them can be used in INCITE attack, and 3 existing n-day vulnerabilities which are also exploitable with the INCITE attack. The vulnerabilities that can be used in INCITE attacks are listed in Table I.

In the current blockchain ecosystem, there are two main factors for INCITE to be a non-negligible attack.

- Some blockchain projects, such as the Bitcoin, may have clients implemented in several languages at the same time. Maintaining software compatibility between them is very challenging, and different clients may behave differently in some corner cases, resulting in the introduction of inconsistencies.
- Some common security bugs can also lead to undefined software behavior, making it vulnerable to INCITE attacks. We will summarize these security vulnerabilities and introduce the exploitation in this section.

Our attack is for victims who actively participate in the consensus process and execute smart contracts. Our attacker (1) "incites" victim nodes to perform the transaction differently by constructing ambiguous transactions. The ambiguous transactions can be constructed by using memory out-of-bounds, uninitialized memory, programming language differences, platform API differences, floating-point precision differences, and so on. The victim nodes refuse to accept blocks from unaffected nodes due to transaction inconsistency, which will lead to (2) network splitting. Finally, the attacker (3) exploits the corrupted state to achieve other attacks, such as the double-spend attack. We now detail each step of our attack.

A. Constructing ambiguous transactions to incite victim nodes

Since the state is not explicitly encoded in transactions, nodes must replay and execute all transactions in the proper order to maintain the local state independently. When performing transactions, any difference between nodes may cause the nodes to lose synchronization and thus get isolated from the normal nodes. We found that some kinds of memory vulnerabilities can be exploited to cause differences between nodes to implement INCITE attacks. Besides, INCITE attacks can also be made by taking advantage of differences in platforms and programming languages.

1) "Trivial" memory vulnerabilities: In typical scenarios, out-of-bounds read and uninitialized memory are generally not sufficient to directly cause severe consequences, since it is challenging for them to achieve control flow hijacking without cooperating with other vulnerabilities, such as out-of-bounds write, and further irreversible damage. However, the attacker only needs to affect the *data flow* through the vulnerability of memory insecurity to interfere with the control flow of the smart contract to achieve the attack, and it does not need to bother with remote code execution.

Out-of-bound access vulnerability. Considering memory randomization mechanisms such as ASLR and stack canary, the memory space of nodes is full of random values, and the memory layout of different nodes varies greatly. Therefore, the value read through out-of-bounds memory reading on different nodes can be regarded as random sources. Furthermore, if a control flow decision depends on a variable that can be controlled by a vulnerability during the transaction, it may cause different execution results on different nodes.

We discovered an out-of-bound vulnerability in EOS (CVE-2018-20690). When EOS WebAssembly virtual machine converts an offset to a memory address, there is an integer overflow which can be exploited to achieve out-of-bound read.

Fig. 3 demonstrates a simplified version of exploitation. In the function "diverge()", `out_of_bound` refers to a variable that can achieve out-of-bound read, which also bypasses the protection of the smart contract virtual machine. In the initial state, the state of the two nodes $(victim_A, victim_B)$ is synchronous. However, two nodes with different memory states may lead to different `oracle`, which will cause inconsistent transaction execution in different nodes.

We also discovered a memory corruption vulnerability (CVE-2018-20689) in EOS, which will eventually lead to control flow hijacking. By hijacking the control flow, an attacker can get different execution paths directly on different nodes.

Uninitialized memory. The value of uninitialized memory depends on historical transactions which have executed on the node. Uninitialized memory on different nodes is more likely to have the save value because of the same executing order. So our attack succeeds if we can alter the uninitialized memory of the target victim node solely. To do this, a feasible way is to issue a transaction which is destined to fail and will not be forwarded to other nodes, and then send it to the victim node. This doomed transaction needs to update the uninitialized

```
function diverge(uint index) public {
 uint oracle = out_of_bound[index];
 if (oracle % 2 == 0) {
    state transition to State_{n+2}
 }
 else {
    state transition to State_{n+3}
 }
}
```

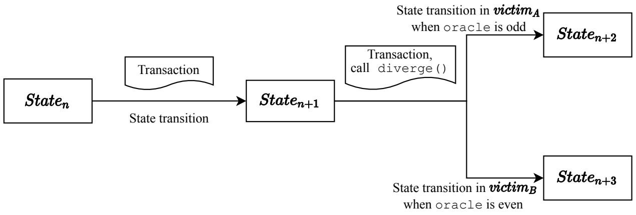

Fig. 3. Sample code vulnerable to INCITE attacks: the upper part shows the vulnerable code, whereas the lower part shows the state change in two different victim nodes, and variable `out_of_bound` refers to a variable that can be accessed beyond the boundary. The two nodes will not be able to reach consensus after the attack.

memory and then trigger a runtime exception actively to exit with an abnormal state. Through this exploiting technique, we can accurately select the target of the attack without causing a large-scale impact, which is conducive to the attacker to covertly attack.

We discovered an uninitialized memory vulnerability in EOS (CVE-2018-20696). With this vulnerability, we can successfully import a random memory value during the transaction execution process to achieve an INCITE attack. More details about this vulnerability are shown in Section V.

2) Different implementations: Clients implemented in *different programming languages* are a potential source of transaction inconsistencies. Not only that, the client may behave differently on different platforms, even if implemented in the same language. We conclude several representative vulnerabilities in the following aspects.

Standard library. The C standard library (libc) is the standard library for the C programming language and is widely used as the cornerstone of various applications. Although the C standard libraries on different platforms are unified, there may still be slight gaps in some APIs. We discovered a standard function implementation inconsistency vulnerability in EOS (CVE-2018-20692). In the definition of ANSI C, the return value of the `memcmp(s1, s2, length)` function is required to be: less than 0, equal to 0, or greater than 0. Some library implementations will only return `-1` or `1` if not identical, while others will return the difference of the first unmatched byte. An attacker can exploit the differences of APIs to construct ambiguous transactions.

Such similar small differences also exist between different languages. The implementation of `BigInteger` divisions in different languages (Python, C#) of NEO are different (CVE-2018-20746), which can be exploited by the attacker to create ambiguous transactions.

Cryptographic algorithm. Differences in the implementation of cryptographic libraries on different platforms and languages are usually trivial, but fatal. ECDSA (Elliptic Curve Digital Signature Algorithm) is a widely used cryptographic algorithm in blockchains to verify the source of data. In order to facilitate the writing and execution of smart contracts, basic operations such as ECDSA are encapsulated into a single instruction for easy calling. However, such a basic algorithm still has slight differences in the implementation of different languages.

The ECDSA standard [15] enforces that the private key should not be zero. Such standard is strictly enforced in multiple cryptographic libraries written by Python and JavaScript. Still, we have found that some ECDSA libraries of Golang allow private key $p = 0$ for signature generation and verification. We have located such a vulnerability in ONT (CVE-2018-20694) along this line of thought, which can cause inconsistent transaction execution by exploiting differences in the implementation of cryptographic libraries. Attackers can utilize this feature to construct smart contracts to distinguish nodes between different language implementations and cause the inconsistent state between different kinds of nodes.

3) Leaky abstraction of APIs: The transaction execution layer should abstract away the irrelevant details of the bottom layer and shield non-deterministic states that are not related to historical transaction records. If the API or instruction set of the virtual machine has not been carefully considered, then the smart contract can distinguish different nodes and choose different execution paths on different nodes during the execution. Some wrong design is apparent, such as allowing smart contracts to obtain the current time, etc. However, all abstractions are more or less leaky, which makes it difficult to find such vulnerabilities without careful audit.

We have discovered a vulnerability related to leaky abstraction (CVE-2018-20693) in NEO. In NEO, the hash of a transaction does not depend on its *witness scripts*, but only on the transaction data. The witness scripts of NEO can be seen as a piece of code to verify transactions, so it is easy to construct two different witness scripts with an equivalent result. It allows the attacker to build two transactions with the same transaction data but different witnesses that both pass the verification. An attacker can broadcast two transactions with the same hash, but with two different well-crafted witness scripts, which leads to the same block hash, and the block would be indistinguishable to victim nodes. However, the attacker can abuse the `Witness_GetVerificationScript` API to distinguish the two witnesses from different transactions, and further perform distinct operations that end up with inconsistent states between the nodes.

Through the analysis of historical vulnerabilities, we also found several vulnerabilities that can be used to implement INCITE attack, but there are currently no reports that these vulnerabilities have been intentionally exploited. CVE-2017-14457 [16] is an out-of-bounds read vulnerability in CPP-Ethereum caused by improper handling for `create2` opcode. An attacker can abuse this vulnerability to implement memory disclosure, and distinguish nodes based on the memory data and further INCITE attacks. As an example of another vulnerability (CVE-2018-20421) in Ethereum, if the memory size of the nodes is different, the execution results may be distinct.

B. Splitting the network

When an ambiguous transaction spreads in the P2P network, different nodes that received the transaction choose their own executing path depending on their internal state. For those blockchains which do not have a field to verify transaction execution, the ambiguous transaction would split the network implicitly. In other words, nodes cannot detect the state differences between each other, even with different account balance statistics.

However, *chain-based* blockchains with the verification field to protect the transaction execution, are still vulnerable to INCITE attacks. The reason is that nodes reject the incoming blocks from other forks due to the failure of the verification phase (As described in Section III). Nodes will encounter numerous block verification errors, but this does not help them defend against INCITE attacks. Furthermore, the nodes on different forks will reorganize themselves into a smaller network and be able to continue their activities, such as generating transaction confirmations.

On the other hand, *BFT-based* blockchains are able to achieve block consensus in the presence of Byzantine or malicious nodes, but do not develop forks during the consensus process. If a node detects itself have different executing paths with an ambiguous transaction, the node will lose the synchronization and cannot process future transactions. However, some public blockchains that we analyzed do not have mechanisms to ensure the consistency of transaction execution, which makes themselves still vulnerable to INCITE attacks.

We demonstrate in Fig. 3 that how to divide the network into two halves, but it is also possible to divide the network into hundreds of pieces with some minor changes.

C. Exploiting the corrupted nodes

In spite of splitting the network into parts, INCITE attacks are also feasible to build other attacks.

Expel mining power. For blockchains based on Proof-of-work, utilizing INCITE attacks can incite victim miners to mine on the wrong view of the blockchain, thereby expelling their mining power, which will make the attacker easier to achieve the 51% attack. INCITE can lightly implement large-scale attacks by taking advantage of implementation differences, without requiring a large amount of mining power or control of the victim's network.

Frame victim nodes. For blockchains based on Proof-of-Stake, INCITE attacks may cause nodes to suffer *stake losses* instead of wasting mining power. Some blockchains (such as Ethereum's Casper) include a punishment mechanism to prohibit that nodes mine conflicting blocks without risking their stake, also known as nothing-at-stake attack [17]. The core is that the nodes involved in the production of the block need to stake a certain amount of assets and be supervised by a series of punishment conditions. When a node produces blocks on both chains at the same time or does not produce blocks at the *latest height*, the node's stake will be slashed.

```
function double_spending(uint index, uint magic_value)
    public {
  uint oracle = uninitialized_memory[index];
  if (oracle == magic_value) {
    Transfer token to the merchant.
  }
  else {
    Do nothing.
  }
}
```

Listing 1. Sample code to achieve double-spending attacks.

When the victim node is incited by INCITE, it will not be synchronized to the latest state, causing the node to generate blocks on the abandoned blockchain view, eventually leading to economic penalties.

0-confirmation double spending. In the 0-confirmation transaction, the customer sends its signed transaction to the merchant as a payment voucher *without* waiting for being included in a block [18]. A 0-confirmation transaction is a form of *fast* payment which is used when it is inappropriate to wait 5-10 minutes (for Bitcoin) for a block confirmation. Common application scenarios are P2P cash, online gambling websites, retail payment systems (BitPay [19], ShowPay [20]), etc.

In order to launch a double-spending attack against the victim merchant, the attacker needs to construct an ambiguous transaction to incite the merchant's node. The sample attack code is shown in the Listing 1.

An uninitialized variable `uninitialized_memory` depends on the previous execution state, so different nodes may have different uninitialized values. When `oracle` is equal to `magic_value`, the node will transfer tokens to merchant. What the attacker wants is that the smart contract only transfer tokens when it runs on the victim node, and not transfer tokens on other nodes.

There are two ways to achieve this target. One of them is to read the uninitialized memory by deploying another contract, and then change the calling parameters of this contract to conform to the state of the victim node. However, the disadvantage of this method is that the uninitialized memory on different nodes has a *high probability* of being equal in practice, which is not conducive for attackers controlling the attack range and impact. Another more practical method is to construct a transaction that will *not be forwarded to other nodes* to modify the uninitialized memory to `magic_number`. Any normally executed transaction will be forwarded to other nodes to maintain consensus, but if an error occurs during the execution of a transaction, the transaction will not be forwarded. By modifying uninitialized memory in a transaction and actively triggering an exception (such as division by zero), an attacker can modify the uninitialized memory of the victim individually.

N-confirmation double spending. Based on 0-confirmation transactions, an attacker can easily implement an N-

```
struct interpreter_interface {
  // Omit scaffolding code here...
  void growMemory(Address old_size, Address new_size) {
    // The newly allocated memory is not cleared.
    current_memory_size += new_size.addr - old_size.addr;
  }
}
```

Listing 2. An uninitialized memory vulnerability in EOS

confirmation double-spending attack. In an N-confirmation transaction, a merchant releases goods only after the transaction is confirmed in a block of depth $N-1$ in the blockchain.

Similar to the double-spending attack with 0-confirmation transactions, the extra work that the attacker needs to do is to incite *multiple* victim nodes (includes the merchant), instead of the merchant node only, to work for the transaction confirmation (such as Proof-of-work). These incited nodes will complete the confirmation of the attacker's transaction on the obsolete view of the blockchain. After this, the attacker can show the confirmation on this obsolete view of blockchain to the victim merchant, and receives the goods. Since only a part of the nodes was incited, most of the remaining nodes will not perform the transfer operation to the merchant, so the attacker did not make a payment on the non-obsolete chain. The incited miners' blockchain is orphaned, and the attacker obtains goods without paying.

V. INCITE ATTACKS ON PRIVATE CHAINS

To further clarify the INCITE attack, we will take EOS as an example to build an EOS private chain and demonstrate our INCITE attack with an uninitialized memory vulnerability (CVE-2018-20696). We will first briefly introduce the basics of the EOS platform, explain the causes of vulnerabilities, outline the steps to build an experimental environment, and finally demonstrate the INCITE attack.

A. Cause of the vulnerability

Nodeos is the core of the EOS platform. Nodeos handles the blockchain data persistence layer, peer-to-peer networking, and contract code scheduling. Since only one transaction is executed at a time, all transactions share the same memory space as heap memory when they are executed.

As shown in Listing 2, this linear memory shared between different contracts will not be cleared when the execution code is switched.

B. Testing environment

We deployed 4 EOS nodeos (*historical version*) on the same host, and connect nodes with P2P connections. Our attack aims to destroy the synchronization of multiple nodes, and further cause *different balances* of the same account in different nodes. To achieve this goal, we built a private chain with 4 producer nodes with names producer_a to producer_d.

```
(func $modify_uninit_memory (param $data i32)
  ;; Adjust the accessible memory limit
  (grow_memory
    ;; Increase the memory limit of 64KiB
    (i32.const 1)
  )
  drop

  ;; Write data to the requested memory
  (i32.store
    (i32.const 0x1fffc)
    (get_local $data)
  )

  ;; Raise an exception
  (call $raise_exception)
)
```

Listing 3. Modify uninitialized memory

TABLE II
STATUS COMPARISON BEFORE AND AFTER THE ATTACK

	producer_a		producer_b		producer_c		producer_d	
	user_a	user_b	user_a	user_b	user_a	user_b	user_a	user_b
Before	10	0	10	0	10	0	10	0
After	10	0	10	0	10	0	**9**	**1**

C. Exploitation

The attack process can be divided into two steps:

1) *Modify the uninitialized memory.* Since the EOS contract uses WebAssembly as its instruction set, we use the WebAssembly code in Listing 3 as an example to demonstrate how to modify the uninitialized memory for subsequent exploits.

Due to the vulnerability, an attacker can write data in the memory allocated by `grow_memory` that will not be cleared, thereby planting *inconsistencies* in the node's memory address space. But a successful executed transaction is executed at all producer nodes to maintain synchronization. In order to avoid unwanted state modification from being broadcast to other nodes, the attacker could actively trigger an exception to make the contract exit with abnormal states.

2) *Read uninitialized memory and transfer money.* As shown in Listing 4, the function `read_memory_and_transfer` accepts a parameter representing the amount of the transferred amount, and determines whether to transfer a certain amount of tokens from user_a to user_b based on the data which is written by the previous contract.

The sequence diagram of the overall attack is shown in Fig. 4. The attack first cause differences in the memory space between nodes through the vulnerability, and then convert the temporary differences in memory to long-term persistent inconsistencies in the data layer with a conditional control instruction, finally different balances of the same account on different nodes. Since user_a's balance at producer_a to producer_c has not changed, which allows the attacker to implement a double-spending attack.

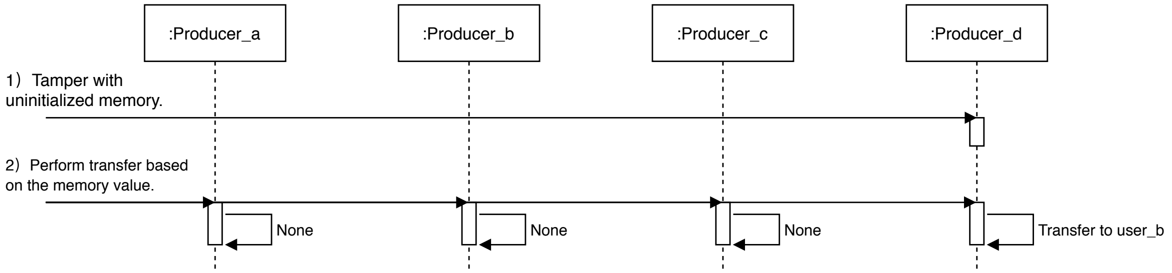
:Producer_a
:Producer_b
:Producer_c
:Producer_d
1) Tamper with uninitialized memory.
2) Perform transfer based on the memory value.
None
None
None
Transfer to user_b

Fig. 4. The sequence diagram of the attack

```
(func $read_memory_and_transfer (param $amount i32)
  ;; Adjust the accessible memory limit
  (grow_memory
    (i32.const 1)
  )
  drop

  ;; Read at the specified offset of the memory.
  (i32.load
    (i32.const 0x1fffc)
  )
  (if
    (i32.eqz)
    (then
      ;; No operation here.
    )
    (else
      ;; Transfer to user_b when the value is not 0.
      (call $transfer_to_user_b
        (get_local $amount)
      )
    )
  )
)
```

Listing 4. Read uninitialized memory and transfer tokens

VI. Defense to the INCITE attacks

A. *Defense mechanism*

A typical design flaw is that only Merkle trees, hash pointers, etc. are recorded in the block header, but state changes caused by transaction execution are not explicitly recorded. This means that nodes rely on independently executing transactions to obtain global status. If the block does not contain verification information about the state, then once attacked by INCITE attacks, the node cannot confirm whether it executed the transaction correctly.

A seemingly effective defense method is that the block producer calculates the hash of the global state after executing all the transactions in a block, and then packs the hash into the block header. In this way, once another node receives a new block, the node first executes all transactions locally and calculates the global state hash after completion. If the state hash obtained locally is the same as the state hash in the block, then no conflict has occurred. If the node finds that the global state obtained by local execution is different from the record in the block, it means that there is an inconsistent execution of a transaction, then the node can reject the block.

However, this is what Ethereum and some blockchains have implemented, and some blockchains that contain transaction execution verification are still vulnerable to INCITE attacks. The reason for this dilemma of chain-based consensus is that even if a node can filter out illegal blocks during the block verification stage, the verification process itself still depends on the state of the node, and the node accepts those blocks or branches that meet their “flavor”. In other words, nodes cannot determine whether this branch is the correct branch without external information, and they may reject one block but accept another wrong block.

For aforementioned BFT-based consensus (Section III-C), due to its absolute certainty, once a node receives a block that meets the cryptographic signature requirements from the network layer, which means that the block must already be included in the blockchain without rollback and forks. If a node finds a block that is incompatible with its state, the node can conclude that an error has occurred in the transaction execution, which can avoid further exploitation on corrupted nodes.

In the existing system, due to neglecting the inconsistencies in the execution of transactions, the current blockchains prefer availability to consistency, which is why the attacker is able to exploit corrupted nodes. To mitigate this obstacle, we need to make a *trade-off* between availability and consistency.

Beyond verifying the execution result of the transaction, the core of defense against INCITE attacks is to reject *all future transactions* when inconsistent transaction execution is found to avoid actual economic losses. Besides, since the global state of the blockchain will increase rapidly over time, if the node needs to hash the entire state every time it generates a block, it will bring unacceptable overhead. Ethereum uses the MPT (Merkle Patricia Tries) data structure to store its state tree to achieve quick verification of transaction execution results. This requires other blockchains to use their unique data structure to save the state makes Ethereum solutions infeasible to other blockchains.

We propose a mitigation solution which is suitable for any data structures, and the key points are as follows.

1) In the block packing phase, the block producer records the write operation sequence $S = \{w_1, w_2, \cdots, w_n\}$ in the transaction execution process of the block, and packs the hash of the sequence $H = hash(S)$ into the block. Considering the complexity of the smart contract, inconsistent execution can be caused by various factors. However, if we can maintain the consistency of the write operation, we can maintain the consistency of the overall state to prevent the post-exploitation after transaction inconsistencies.

2) After the node receives the new block and completes the integrity check, if the block is the latest block of the current chain, it executes the transaction in the virtual machine and records the write-sequence of these transactions locally, and then calculates the hash of the write-sequence, H'.
3) If $H \neq H'$, it means that there is an inconsistency between this node and the node that generated the block when executing the transaction. But this block may be constructed by the INCITE attack or tampering with the state hash during mining, so we only mark the branch where the block located as a state verification failure.
4) In the follow-up, the node needs to monitor the branch marked as failed for verification. If the length of the branch exceeds N blocks of the current branch, the nodes refuse further transactions and prompt users that there is a risk of asset loss to prevent users from actually sending the goods.

   In the defense process, the smaller the N, the earlier the INCITE attack will be discovered, but it is more susceptible to DDoS attacks from other nodes.

It is worth noting that we are not defending by merging hard forks, but by detecting post-exploitation after discovering inconsistencies to refuse to confirm transactions to avoid losses. In other words, the attacker can still make a hard fork, but will not be able to make a double-spending attack. We think there is no effective algorithm to maintain consistency without actually fixing the vulnerabilities.

B. Experiments

As shown in Table III, we implemented our defenses on EOS and tested 5 different kinds of vulnerabilities. The result shows that our mitigation can prevent the post-exploitation after transaction inconsistencies.

We have tested the performance of our defense in different block sizes, and each block contains different number of transactions. Our reinforced client and original one are evaluated in two dimensions, which are the computational overhead and the memory overhead.

As shown in Fig. 5 and 6, the x-axis represents the number of transactions contained in a block, while y-axis represents the additional computation time (Δt) and the additional memory usage (Δm) respectively. From the Fig. 5 and 6, we can see that as the number of transactions increases, both computational overhead and memory overhead increase linearly. This is because as the number of transactions increases, the number of write operations increases accordingly. Fig. 7 shows that the overall performance overhead in computation is about 3.8%, which is acceptable. As for storage usage, which are not demonstrated in the figure, the overall storage consumption is nearly the same as the modification because only the hash of the write-sequences will be written in the block.

TABLE III
EXPERIMENT RESULTS OF OUR DEFENSE

CVE ID	Projects	Descriptions	Results
CVE-2018-20690	EOS	Out-of-bound read	Reject
CVE-2018-20689	EOS	Memory corruption	Reject
CVE-2018-20696	EOS	Uninitialized memory	Reject
CVE-2018-20692	EOS	Inconsistent version implementation	Reject
CVE-2018-14439	EOS	Different implementation in languages	Reject

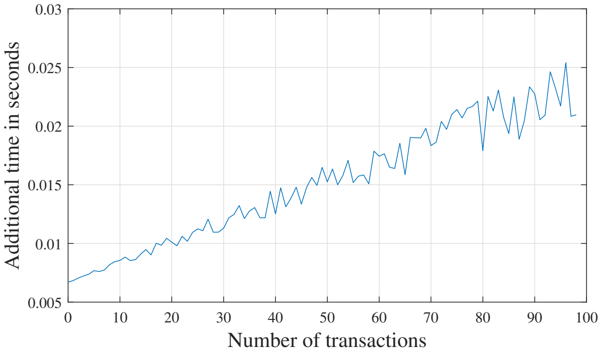

Fig. 5. Computational overhead

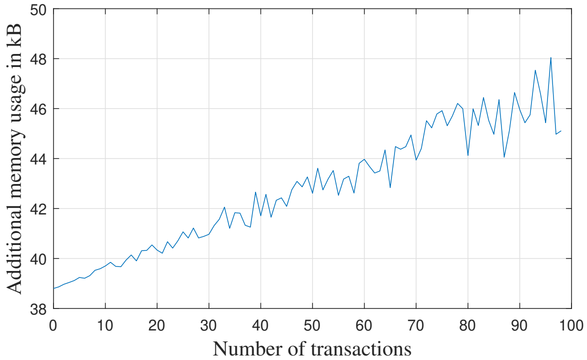

Fig. 6. Memory overhead.

VII. Related Work

Unlike the INCITE attack, which will cause long-range and persistent inconsistencies between nodes, current related work exploit temporary inconsistencies between nodes to implement double-spend attacks. Nodes attacked by such temporary inconsistencies can synchronize with other nodes and

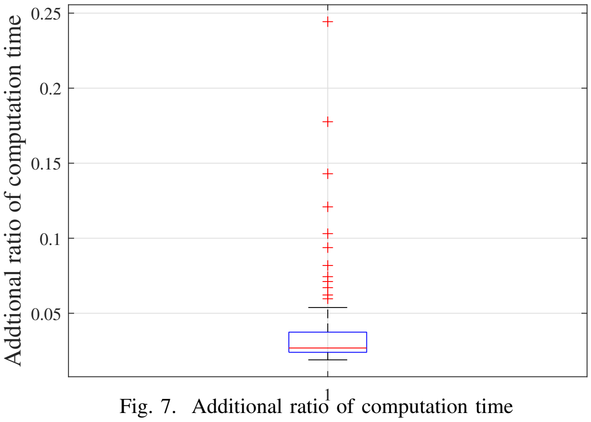

Fig. 7. Additional ratio of computation time

eliminate inconsistencies after the attack ends. In the existing work, temporary inconsistencies are mainly constructed by the following methods.

Attacks against the consensus mechanism. Of the double-spending attacks against the proof-of-work mechanism, the 51% attack is the most widely known. The proof-of-work mechanism relies on the assumption that over half of the computing power of the entire network is honest, and 51% of attacks are achieved by breaking this prerequisite. In this attack, the attacker needs to control over half of the computing power of the entire network to accomplish the attack. Since the attacker has an advantage in the computing power of the whole system, it can control the growth of the blockchain as desired. For example, the attacker can prevent the transaction from getting confirmation during the attack, making it impossible for some or all users to use Bitcoin to make payments. This kind of attack is different from the INCITE attack because the consistency of the public ledger can be restored after a double-spending attack.

Attacks against the P2P networks. If an attacker controls the P2P network that the blockchain system depends on, it can manipulate network packets to construct inconsistencies between nodes. Since the network bandwidth of each participant node is limited, each node only maintains a certain number of TCP connections to connect to the network. If the victim is only connected to the nodes controlled by the attacker, the attacker can fully control the victim's view of the blockchain. When the attacker isolates the victim node, it can consume separately on the controlled node and the uncontrolled node. This inconsistency will be cleared after the victim reconnects to the network, and the victim will realize that they have been attacked after synchronizing with other nodes.

Typical examples include eclipse attacks [11], DNS hijacking attacks, and BGP hijacking attacks [12].

VIII. Conclusions

In this work, we have presented how attackers launch double-spending of cryptocurrencies on blockchains through INCITE attacks. The core of INCITE attacks is to construct an ambiguous transaction and broadcast it to the network, causes different execution results of smart contracts in different nodes, thereby misleading the victim nodes to a maliciously constructed state.

We implemented INCITE attacks by exploiting some "trivial" memory vulnerabilities and implementing differences. We demonstrated our attacks on the actual mock blockchain of EOS, proving that INCITE attacks are able to cause a severe impact on the real environment. Moreover, we discovered 7 zero-day vulnerabilities and evaluated 3 n-day vulnerabilities on five popular blockchain projects as real-world demos. A possible defense measure against INCITE attacks is also proposed and tested.

Acknowledgement

This project is supported in part by National Natural Science Foundation of China 61772308, 61972224 and U1736209, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In Jan Camenisch and Doğan Kesdoğan, editors, *Open Problems in Network Security*, pages 112–125, Cham, 2016. Springer International Publishing.

[2] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[3] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 3:1 – 29, 2019.

[4] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[5] EOS.IO technical white paper v2. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md, April 2018.

[6] NEO a distributed network for the smart economy. https://github.com/neo-project/docs/blob/master/docs/en-us/basic/whitepaper.md, July 2017.

[7] A new high-performance public multi-chain project & a distributed trust collaboration platform. https://ont.io/wp/Ontology-Introductory-White-Paper-EN.pdf, July 2019.

[8] Security alert: Consensus bug in geth v1.4.19 and v1.5.2. https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/, November 2016.

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

[10] Martijn Bastiaan. Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin. In *Availab le at http://referaat. cs. utwente. nl/conference/22/paper/7473/preventingthe-51-attack-astochastic-analysis-of-two-phase-proof-of-work-in-bitcoin. pdf*, 2015.

[11] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, Washington, D.C., August 2015. USENIX Association.

[12] M. Apostolaki, A. Zohar, and L. Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 375–392, 2017.

[13] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[14] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[15] Daniel RL Brown. Sec 1: Elliptic curve cryptography. *Certicom Research*, page v2, 2009.

[16] CVE-2017-14457. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14457, November 3 2018.

[17] Wenting Li, Sébastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 297–315, Cham, 2017. Springer International Publishing.

[18] Ghassan O. Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 906–917, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Bitpay, accept bitcoin for online payments. https://bitpay.com/, June 2020.

[20] Showpay, make bsv to be global daily cash. https://showpay.io/, June 2020.