Analysis of CVE-2018-17144

Team Cointurkeys: Megha Hegde, Abby Chou, Josiane Uwumukiza

I. Introduction and Background

In 2018, a vulnerability (CVE-2018-17144) was disclosed that allowed miners to create bitcoin out of thin air by doublespending outputs within one transaction. The issue was first introduced in Bitcoin Core 0.14 through an optimization pull request (PR #9049) that simplified a seemingly redundant validation check ensuring that no transaction spent the same input twice. Although the UTXO update logic still contained limited sanity checks, it lacked full error handling. Therefore, attempts to double-spend an input inside a transaction could crash the affected nodes. The vulnerability worsened in Bitcoin Core 0.15.x, 0.16.0,0.16.1, and 0.16.2. A redesign in the UTXO tracking system changed a key assertion: instead of checking if an output was previously unspent, it only asserted that it existed. This small change meant that the system wouldn't detect when a miner spent an output twice in the same new transaction, enabling a malicious miner to inflate Bitcoin's supply by double-spending. This made this vulnerability one of the most severe ever encountered in Bitcoin Core, as it threatened the fundamental 21 million coin limit.

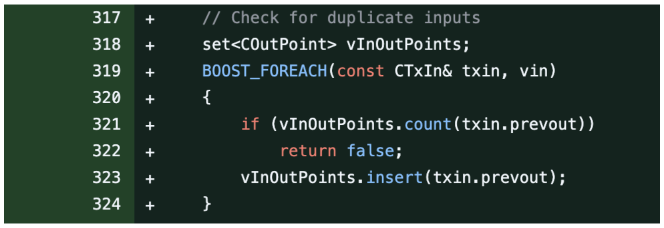
It wasn't until well after rolling out the aforementioned versions of Bitcoin that an anonymous community member reported the potential bugs that the removal of this check caused. After the bug was initially reported to bitcoin developers on Sept 18, 2018, developers published version 0.14's DDOS vulnerability that would allow a doublespending transaction to crash nodes, released a patch, and urged nodes to quickly update their software. It wasn't until after over half the network had upgraded that the developers released a full disclosure of the vulnerability on Sept 20, 2018, including how it could have been exploited to doublespend bitcoin in later versions. In doing so, they were able to reduce the risk of the inflation vulnerability being exploited, since the network had already upgraded to the patched version by the time the full extent of the bug was publicized.

If the bug had been discovered first by a malicious user and exploited, the potential impact of it is debated and very uncertain. If the bitcoin developers were willing to roll back the adversarial transactions with a hard fork like in the DAO incident, then there would be no economic impact. But if they weren't willing to (which seems very plausible), it could have been disastrous, which is why the vulnerability received widespread attention. It also resurfaced conversations about the dangers of having most of the network running the same bitcoin client software, as a “small” bug like this could allow disastrous transactions to be accepted by the entire network before anyone would notice.

II. Technical Details

A. PR 443: The Original Check

In July 2011, PR 443 introduced a check ensuring that a transaction didn’t doublespend the same output. The check was added to CheckTransaction, a suite of basic sanity checks used to immediately remove any obviously invalid transactions. For example, it checks that transactions don’t have null inputs or negative or overflowing outputs, and, as of PR 443, don’t doublespend the same UTXO. It’s not a comprehensive validity check–for that, you need the context of the entire blockchain–but it does help immediately flag obviously invalid transactions. CheckTransaction is run in two different scenarios: it’s run by mining nodes before admitting a transaction into the mempool, and by validating nodes when validating transactions.



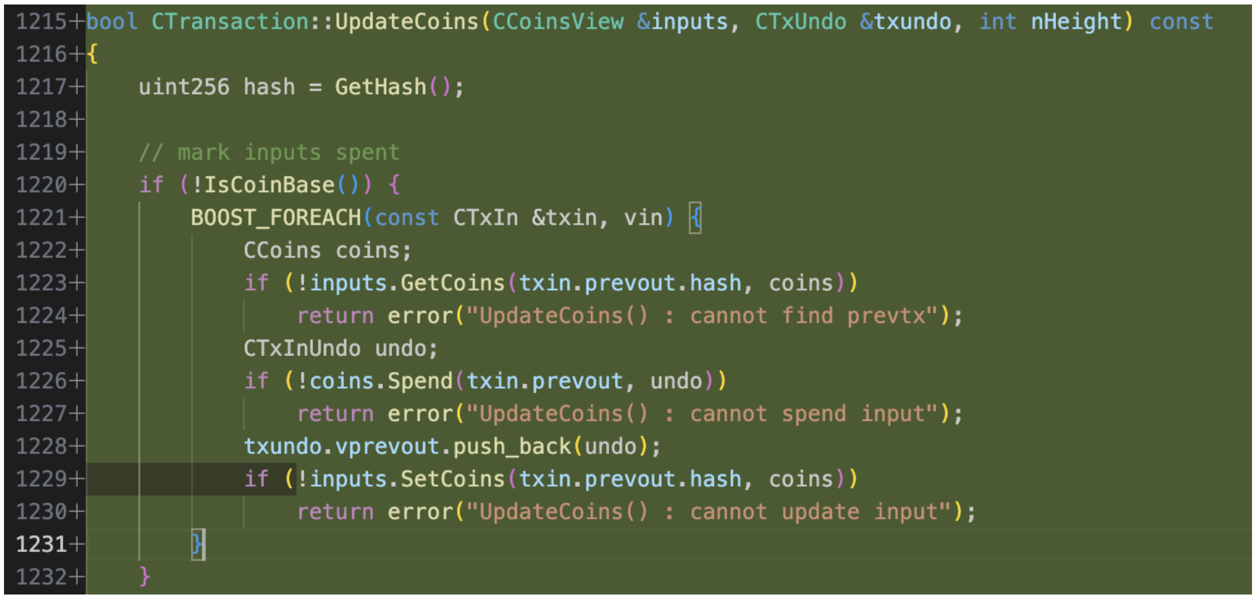
```
+     // Check for duplicate inputs
+     set<COutPoint> vInOutPoints;
+     BOOST_FOREACH(const CTxIn& txin, vin)
+     {
+         if (vInOutPoints.count(txin.prevout))
+             return false;
+         vInOutPoints.insert(txin.prevout);
+     }
```

Code Change for PR 443

The check added in PR 443 was intentionally redundant. If a transaction doublespent the same output, it would also be caught by ConnectInputs, the function that walks through the chain and verifies that each output is spent at most once. The additional check in CheckTransaction was a defense-in-depth measure to fail clearly invalid transactions early in their lifecycle.

B. PR 1677: Ultraprune

In October of 2012, Ultraprune was merged, a huge PR that restructured the way Bitcoin nodes kept track of the UTXO set. Instead of storing the entire transaction history and UTXOs together in one dataset, Pieter Wuille separated them out into a transaction history database and separate UTXO database, cutting down on required storage space significantly. This is when UpdateCoins, the function in question, was first created.



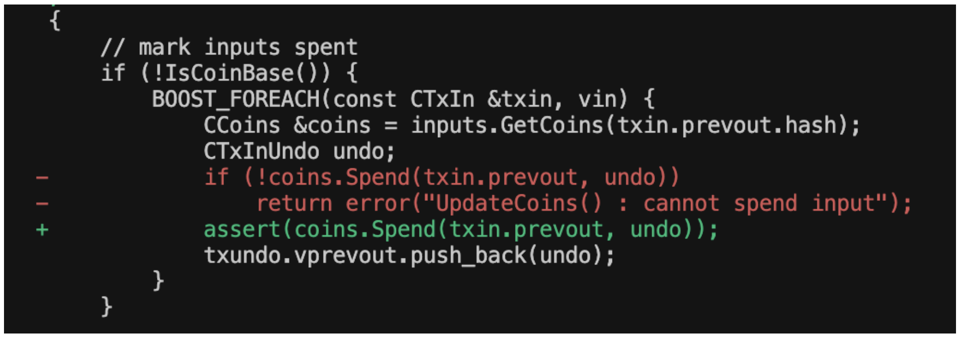
```
bool CTransaction::UpdateCoins(CCoinsView &inputs, CTxUndo &txundo, int nHeight) const
{
    uint256 hash = GetHash();

    // mark inputs spent
    if (!IsCoinBase()) {
        BOOST_FOREACH(const CTxIn &txin, vin) {
            CCoins coins;
            if (!inputs.GetCoins(txin.prevout.hash, coins))
                return error("UpdateCoins() : cannot find prevtx");
            CTxInUndo undo;
            if (!coins.Spend(txin.prevout, undo))
                return error("UpdateCoins() : cannot spend input");
            txundo.vprevout.push_back(undo);
            if (!inputs.SetCoins(txin.prevout.hash, coins))
                return error("UpdateCoins() : cannot update input");
        }
    }
```

UpdateCoins Original Input Checking

If any of the inputs in the transaction is a doublespend, coins.Spend would return false, causing an error that rejects the transaction but does not crash the node.

C. PR 2224: An Unassuming Change

In January of 2013, Pieter Wuille merged PR 2224, another large PR that introduced a new data type for cleaner error handling. However, in the midst of this expansive PR was this tiny, unassuming change:



```
 {
     // mark inputs spent
     if (!IsCoinBase()) {
         BOOST_FOREACH(const CTxIn &txin, vin) {
             CCoins &coins = inputs.GetCoins(txin.prevout.hash);
             CTxInUndo undo;
-            if (!coins.Spend(txin.prevout, undo))
-                return error("UpdateCoins() : cannot spend input");
+            assert(coins.Spend(txin.prevout, undo));
             txundo.vprevout.push_back(undo);
         }
     }
```

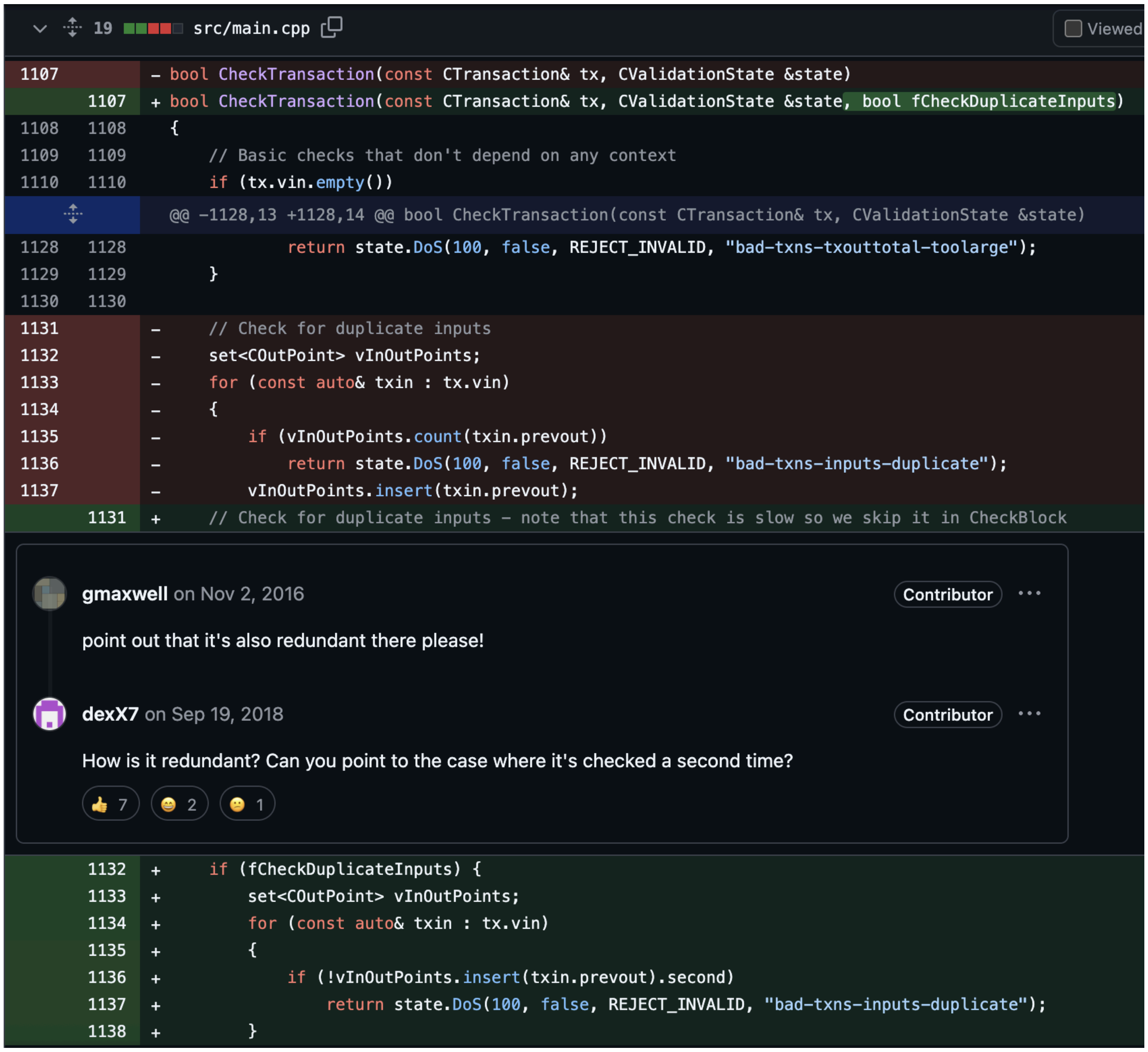
The relevant change from PR 2224

The discussion around the PR doesn't mention the change, so it seems as though it flew relatively the radar of the other members of the bitcoin dev community. At the time, it made sense to assert that the coin was spendable, since all the other consensus checks would have already made sure that the transaction was valid well before reaching the UpdateCoins function. If the assert failed, it couldn't be from a consensus error, but from some memory corruption, in which case the desired behavior is that the node would shut down.

However, the issue with 2224 is that the other consensus checks don't check against a UTXO being spent twice in the same transaction: only the check introduced in PR 443, in CheckTransaction, checks for this odd edge case. Thus, PR 2224 unknowingly made that small check in CheckTransaction a mission-critical check, no longer redundant.

D. PR 9049: DoS Vulnerability

In November of 2016, the developers, not realizing that PR 2224 had made the duplicate input check in CheckTransaction critical, merged PR 9049 which partially removed it. Specifically, PR 9049 added a flag to the CheckTransaction function that would allow the input check to be included or excluded, a flag that would be on (running the check) when miners admit transactions to the mempool, and off (skipping the check) when validators validate blocks.



```
- bool CheckTransaction(const CTransaction& tx, CValidationState &state)
+ bool CheckTransaction(const CTransaction& tx, CValidationState &state, bool fCheckDuplicateInputs)
  {
      // Basic checks that don't depend on any context
      if (tx.vin.empty())
@@ -1128,13 +1128,14 @@ bool CheckTransaction(const CTransaction& tx, CValidationState &state)
              return state.DoS(100, false, REJECT_INVALID, "bad-txns-txouttotal-toolarge");
      }

-     // Check for duplicate inputs
-     set<COutPoint> vInOutPoints;
-     for (const auto& txin : tx.vin)
-     {
-         if (vInOutPoints.count(txin.prevout))
-             return state.DoS(100, false, REJECT_INVALID, "bad-txns-inputs-duplicate");
-         vInOutPoints.insert(txin.prevout);
+     // Check for duplicate inputs - note that this check is slow so we skip it in CheckBlock
```

gmaxwell on Nov 2, 2016 — Contributor

point out that it's also redundant there please!

dexX7 on Sep 19, 2018 — Contributor

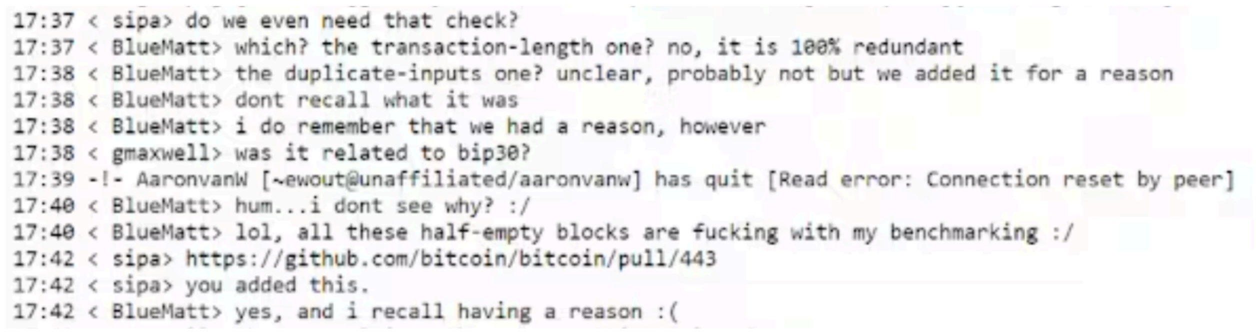
How is it redundant? Can you point to the case where it's checked a second time?

👍 7 😄 2 😕 1

```
+     if (fCheckDuplicateInputs) {
+         set<COutPoint> vInOutPoints;
+         for (const auto& txin : tx.vin)
+         {
+             if (!vInOutPoints.insert(txin.prevout).second)
+                 return state.DoS(100, false, REJECT_INVALID, "bad-txns-inputs-duplicate");
+         }
```

Diff from PR 9049 making the duplicate input check optional

The developers conversed about why the check had originally been added, but Matt Corallo, the original author of PR 443, couldn't remember why it was necessary back then, and the PR wasn't closely examined.



```
17:37 < sipa> do we even need that check?
17:37 < BlueMatt> which? the transaction-length one? no, it is 100% redundant
17:38 < BlueMatt> the duplicate-inputs one? unclear, probably not but we added it for a reason
17:38 < BlueMatt> dont recall what it was
17:38 < BlueMatt> i do remember that we had a reason, however
17:38 < gmaxwell> was it related to bip30?
17:39 -!- AaronvanW [~ewout@unaffiliated/aaronvanw] has quit [Read error: Connection reset by peer]
17:40 < BlueMatt> hum...i dont see why? :/
17:40 < BlueMatt> lol, all these half-empty blocks are fucking with my benchmarking :/
17:42 < sipa> https://github.com/bitcoin/bitcoin/pull/443
17:42 < sipa> you added this.
17:42 < BlueMatt> yes, and i recall having a reason :(
```

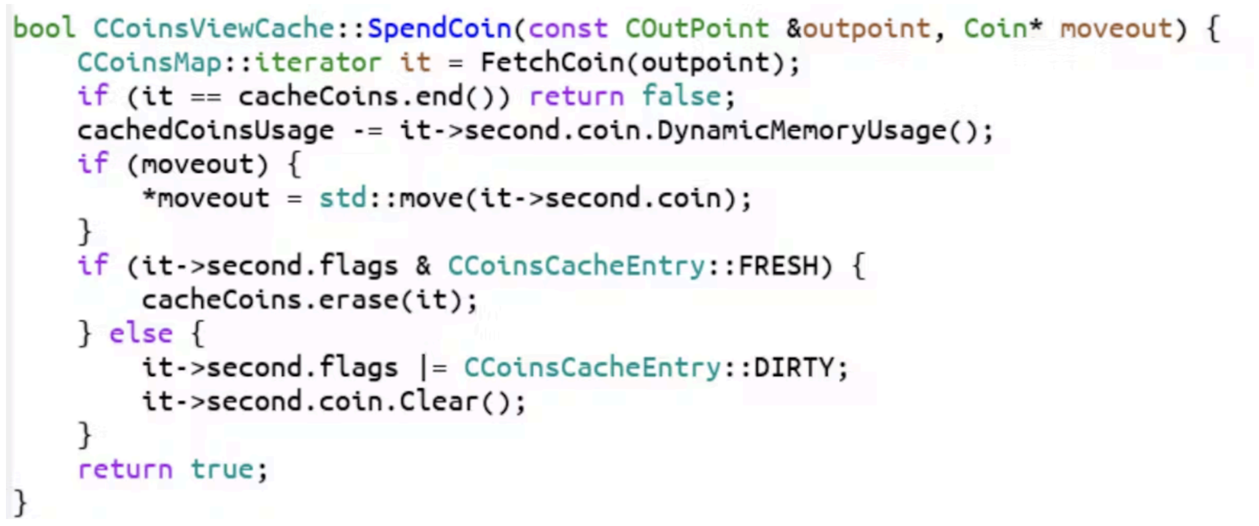
Conversations between developers Pieter Wuille and Matt Corallo about PR 9049

Without this check, validators wouldn't catch a transaction that spent the same input twice in the same transaction, but the assert from PR 2224 would fail, crashing the node. This creates a DoS vulnerability where an attacker can crash nodes by sending them blocks that doublespend an input in the same transaction. However, such a transaction could not be included in blocks produced by miners, since the duplicate input check is still run in CheckTransaction before admitting transactions to the mempool. Thus, to exploit the vulnerability, an attacker would need to create and mine their own adversarial block, an extra cost that may be part of the reason this vulnerability was never exploited.

E. UTXO Changes in v 0.15, and the Inflation Bug

In v.015, a series of changes to the way UTXOs are stored during the verification process made the issue even worse. In all previous versions of Bitcoin core, as soon as a UTXO was seen in the blockchain it was stored, with a "spent" flag to indicate whether it had been spent. Thus, if the node had previously seen the UTXO in the blockchain, it would be stored in its memory.

However, in v.015, the developers decided to optimize storage space by removing UTXOs from cache memory after being consumed. The resulting behavior was that if a UTXO was in cache with the "FRESH" flag set, typically meaning that the UTXO had been created in the same block, and then it was doublespent in the same transaction, it would trigger the aforementioned assert and crash the node. But if a UTXO was in the cache with the "DIRTY" flag set, meaning that it had been loaded from disk (i.e. had been created in a previous block, not this one), and then that UTXO was spent, it would be cleared from the cache entirely, and if it was doublespent in the same transaction, the transaction would go through with no problem, causing inflation.



```
bool CCoinsViewCache::SpendCoin(const COutPoint &outpoint, Coin* moveout) {
    CCoinsMap::iterator it = FetchCoin(outpoint);
    if (it == cacheCoins.end()) return false;
    cachedCoinsUsage -= it->second.coin.DynamicMemoryUsage();
    if (moveout) {
        *moveout = std::move(it->second.coin);
    }
    if (it->second.flags & CCoinsCacheEntry::FRESH) {
        cacheCoins.erase(it);
    } else {
        it->second.flags |= CCoinsCacheEntry::DIRTY;
        it->second.coin.Clear();
    }
    return true;
}
```

Relevant SpendCoin code clearing UTXOs from the cache

It’s clear that when developing these changes, no attention was paid to the behavior of the node with a doublespent input in the same transaction, as this was likely seen as an edge case handled by other code. However, the CheckTransaction code that should have handled it didn’t, and there also happened to be no unit tests covering this case.

III. IMPLICATIONS

A. Denial of Service Vulnerability

The DoS bug present in Bitcoin Core versions 0.14-0.14.2 allowed a miner to construct a valid block containing a transaction that double-spent a single UTXO within the same transaction. When this block was validated, affected nodes would crash. Critically, the block itself still required valid proof-of-work, which meant the attacker had to expend computational resources equivalent to mining a legitimate block.

Economically, there is a clear incentive misalignment: the attack required sacrificing the block reward (12.5 BTC at the time) in exchange for causing node instability. Since the block would not propagate reliably across all clients (only clients running bitcoin core 0.14), the attacker would likely forfeit the ability to fully collect the reward. Thus, the strategy yields a purely negative expected monetary payoff, making it unattractive for profit-driven adversaries.

The only potential benefit arises from network disruption, not financial gain. However, short-lived node crashes do not meaningfully affect consensus formation or long-run block production. Rational miners, whose incentives are structurally bound to long-term network viability and predictable reward streams, therefore face a strong disincentive to deploy this attack. The vulnerability presents theoretical systemic risk, but its equilibrium outcome within Bitcoin's incentive structure implies a self-limiting attack vector: only actors seeking to incur a cost for non-economic reasons would consider using it, such as governments or large organizations who have some political motive to discredit or destabilize bitcoin.

B. Inflation Vulnerability

The inflation bug in versions 0.15–0.16.2 was fundamentally different from the DoS vulnerability because it introduced the possibility of violating Bitcoin's fixed monetary supply rule by allowing a transaction to spend the same UTXO multiple times *without* crashing the node. Any acceptance of such a block would create valid-looking, yet economically illegitimate bitcoin. This attacks the foundation of Bitcoin's value proposition: deterministic scarcity.

From an economic standpoint, the existence of even a feasible inflation attack shifts expected value. Bitcoin's pricing depends on credible commitment to issuance constraints. A break in that constraint, whether temporary or reversed late, introduces institutional risk analogous to a banking system with uncertain reserve backing. However, although the vulnerability could, in principle, mint arbitrary bitcoin amounts, exploiting it profitably would still require economic feasibility under real market conditions.

Any inflated block would create a visible anomaly on public ledgers, immediately triggering protocol-level detection by diverse client implementations. Competing miners would likely reject and mine on top of an alternate chain, creating a split in which rational miners converge on the non-inflationary chain since its long-term stability preserves future rewards. Moreover, social consensus would almost certainly coordinate a rollback or orphaning of the inflationary block. Therefore, the attacker pays for the block's proof-of-work and risks losing the entire reward with no guarantee of retaining illicit coins.

Thus, while inflation threatens Bitcoin's core monetary invariant, the equilibrium incentive structure implies that the expected payoff of a unilateral exploitation is negative for economically rational agents. The only actors for whom the attack has positive expected value are those seeking to degrade network credibility rather than extract profit, e.g., state-level or ideological adversaries.

IV. CONCLUSION

A. Recommendations

Based on the analysis of CVE-2018-17144, several concrete engineering and process improvements emerge as necessary safeguards for consensus-critical systems. First, duplicate and defense-in-depth validation must be treated as a security requirement rather than a performance cost. Redundant checks, even when theoretically “unnecessary,” provide critical protection against emergent bugs created by later refactors.

Second, performance optimizations in financial or consensus software should be gated behind formal verification, adversarial testing, and stress testing, rather than micro-benchmark gains alone. Optimization that weakens correctness guarantees must be rejected by design.

Third, long-lived codebases must explicitly document and continuously revalidate invariants and assumptions. In this incident, later changes unknowingly relied on assumptions established by earlier code. Systematically recording these dependencies would reduce the risk of future latent vulnerabilities.

Fourth, systems like bitcoin need more comprehensive testing. If there was simply a unit test including a transaction that doublespent the same UTXO, then it would have been immediately clear when this security was violated. As it happened, the developers forgot about this edge case when making changes to the code, but a test case could have encoded this edge case in permanent working memory.

Finally, decentralized systems require improved mechanisms for rapidly coordinating security upgrades. While Bitcoin’s response was fast by decentralized standards, the long tail of vulnerable nodes demonstrates a structural weakness in patch propagation that should be addressed through better alerting, version signaling, and upgrade incentives.

B. Key Takeaways

CVE-2018-17144 demonstrates that catastrophic failures in distributed systems often arise not from single errors, but from the interaction of reasonable decisions made over long periods of time. Each individual change that contributed to this bug was reviewed, approved, and justified in isolation, yet their composition silently removed critical safety guarantees.

The incident reinforces that in consensus-critical software, correctness must always dominate performance. Redundancy is not waste; it is resilience. Assumptions must be treated as liabilities that require continual validation, not as permanent truths.

Perhaps most importantly, this near-failure illustrates that the absence of disaster does not imply the presence of safety. Bitcoin survived this vulnerability due to coordination, fast response, and significant luck, rather than perfect engineering. Future systems cannot rely on luck as a security model. The primary lesson is clear: systems securing billions of dollars must be designed under the assumption that every latent bug will eventually be triggered.