BAD PROOFS IN FORMAL VERIFICATION
Critical Bug Payout Report

Jeff Wu
07 Jan 2022

Notional received a critical bug report from a whitehat hacker last night. The Notional team disabled the affected code in under an hour after it was reported. No user funds have been lost, and none are at risk. No user-facing functionality is affected. Users can continue to use Notional to lend and borrow at fixed rates safely. We have created a fix for this issue, and will deploy the change once our auditors have reviewed and confirmed it. Notional Finance Incorporated will pay the full $1 million bounty listed through Immunefi + a bonus of 100,000 NOTE.

Security remains our highest priority and we will continue to offer the top prize of $1 million via Immunefi for critical issues found. We submitted our code for audit by three independent audit firms. The affected code was present in all three of these audits. We also submitted our code for formal verification. The affected code was subject to a formal verification check explicitly designed to detect this particular vulnerability, but due to human error the check was not properly constructed and did not function as intended. Certora will issue a report detailing why that check did not work as intended in the coming days.
LECTURE ROADMAP

What are proofs in Formal Verification

Types of bad proofs

How to tell if a proof is bad

Real life example
THE FORMAL VERIFICATION PROCESS

Proofs that every behavior meets the spec

Unknown Timeout

A HARD TO FIND behavior which violates the invariants

Invariants

Code

Certora Prover
**SIMPLE EXAMPLE**

**SOLIDITY CODE**

```
transfer (address from, address to, uint256 amount) {
    require (balances[from] ≥ amount);
    balancesFrom := balances[from] - amount;
    balancesTo := balances[to] + amount;
    balances[from] := balancesFrom;
    balances[to] := balancesTo;
}
```

**INVARIANT**

```
totalSupply = \sum_{a: address} balances[a]
```

**BUG**

```
from="Alice"
to="Alice"
amount = 18
old.balances(Alice) = 20
new.balances(Alice) = 38
```
SIMPLE EXAMPLE

SOLIDITY CODE

```solidity
transfer (address from, address to, uint256 amount) {
    require (balances[from] ≥ amount);
    balances[from] := balances[from] - amount;
    balances[to] := balances[to] + amount;
}
```

INVARIANT

totalSupply = \[ \sum_{a: \text{address}} \text{balances}[a] \]

PROOF

\[ \sum_{a: \text{address}} \text{old.balances}[a] = \sum_{a: \text{address}} \text{new.balances}[a] \]
ADVANTAGES OF FORMAL VERIFICATION

Exhaustive
- Finds easy to miss bugs

Concrete counter examples
- Found bugs are verifiable

Proofs of correctness
- Hard to verify
- May be misleading!
Certora Verification Language (CVL)

```
rule checkTransfer(address bob, uint256 amount) {
    env e; /* calling context (msg.sender, block.timestamp, ...) */
    uint256 balanceBefore = balanceOf(bob);
    transfer(e, bob, amount);
    assert balanceOf(bob) == balanceBefore + amount;
}
```
PROPERTY - VISUALIZATION OF WANTED BEHAVIOR

Space Of Possibilities

Start State
(Constraint)

Desired State
(assert expression)
PROPERTY - A VIOLATED RULE

Space Of Possibilities

Start State
(Constraint)

Desired State
(assert expression)
FALSE STATEMENTS

LOGIC DEFINES A FALSE STATEMENT AS THE EXISTENCE OF COUNTER EXAMPLE TO A CLAIM
What are proofs in Formal Verification

Types of bad proofs

How to tell if a proof is bad

Real life example
Vacuous:
- Empty
- Meaningless
- Lacking of significance
- Lacking contents which could or should be present
URI KIRSTEIN - 29 years old, don’t have any children.

Statement - If I let my children drink Colombian coffee, they will sleep better.

Logic

Given that I have no children, any statement about them is indisputable.

TRUE
(Vacuous)
Uri Kirstein - 29 years old, don’t have any children.

Statement
If I let my children drink Colombian coffee, they will sleep better

TRUE (Vacuous)

Statement
If I let my children drink Colombian coffee, they will not sleep at night

TRUE (Vacuous)
PROPERTY - VISUALIZATION OF WANTED BEHAVIOR

Space Of Possibilities

There are no starting states

Desired State
(asser expression)
OpenZeppelin ERC1155 function

function balanceOf(
    address account,
    uint256 id)
public view virtual
override returns
(uint256)
{
    require
    (account != address(0),
    "ERC1155: address zero is not a
valid owner");
    return
    _balances[id][account];
}

Certora Verification Language (CVL)

// If the user has a token, then the token should exist
rule held_tokens_should_exist {
    address user;
    uint256 token:
    require balanceOf(0, token) == 0;
    // This assumption was proven in a separate rule
    require balanceOf (user, token) <= totalSupplyOf (token);
    assert balanceOf (user, token) > 0 => token_exists (token);
Certora Verification Language (CVL)

// If the user has a token, then the token should exist
rule held_tokens_should_exist {
    address user;
    uint256 token:
    require balanceOf(0, token) == 0;

    assert 0 > 1;
“our experience has shown that typically **20% of specifications pass vacuously** during the first formal-verification runs of a new hardware design, and that **vacuous passes always point to a real problem** in either the design or its specification or environment”

REACHABILITY CHECK

CVL
Discovering unreachability by adding assert false at the end of the rule

// If the rule passes, then it is vacuous
rule held_tokens should exist vacuity check {
    address user;
    uint256 token;
    require balanceOf(0, token) == 0;

    // This assumption was proven in a separate rule
    require balanceOf(user, token) <= totalSupplyOf(token);
    assert balanceOf(user, token) > 0 => token exists (token);
    assert false;

We expect the rule to reach the assert false at the end and fail
DISJOINT PRECONDITIONS – UNREACHABILITY VISUALIZATION

Space Of Possibilities

- $X < Y$
- $Y < Z$
- $Z < X$

Start State

Desired State (assert expression)
VACUOUS ASSERTIONS – TAUTOLOGY DEFINITION

Vacuous assertions:

- The saying of the same thing twice in different words
- A propositional statement that is always true
- A formula or assertion that is true in every possible interpretation
rule something_is_always_transferred {
    address recipient;
    uint256 balance_before_transfer = balanceOf(recipient);
    require balanceOf(recipient) == 0;

    uint256 amount;
    require amount > 0;

    transfer(recipient, amount);

    uint256 balance_after_transfer = balanceOf(recipient);
    assert balanceOf(recipient) <= balance_after_transfer;
}
TAUTOLOGY VISUALIZATION

Space Of Possibilities

Start State
(Constraint)

Desired State
(assert expression)
FINDING TAUTOLOGIES

Remove all preconditions and the operations, then check if the rule still passes

```plaintext
rule something_is_always_transferred_vacuity_check {
    uint256 balance_after_transfer = balanceOf(recipient);
    assert balanceOf(recipient) <= balance_after_transfer;
}
```
LECTURE ROADMAP

- What are proofs in Formal Verification
- Types of bad proofs
- How to tell if a proof is bad
- Real life example
**Invariant**

- Always the same
- Never changing
- A logical assertion that is always held to be true
- A property which remains unchanged after operations or transformations of a certain type are applied
PROOF BY INDUCTION

1. The base case - after constructor

2. The step - any external/public function
   a. Assume the invariant
   b. Call the function
   c. Check if the invariant is still true
TAUTOLOGICAL INVARIANT

A non-zero asset cannot be both bitmap and active

// BAD INVARIANT
assert 0 <= i && i < 9 &&
getBitmapCurrency(account) != 0 &&
(
  // When a bitmap is enabled it can only have currency masks
  // in the active currencies bytes
  hasCurrencyMask(account, i) &&
  getActiveUnmasked(account, i) == 0 ||
  getActiveMasked(account, i) == 0
)
=> getActiveUnmasked(account, i) != getBitmapCurrency(account)
A non-zero asset **cannot** be both **bitmap** and **active**

```
// BAD INVARIANT
assert 0 <= i && i < 9 &&
getBitmapCurrency(account) != 0 &&
(
    // When a bitmap is enabled it can only have currency masks
    // in the active currencies bytes
    (hasCurrencyMask(account, i) &&
    getActiveUnmasked(account, i) == 0) ||
    getActiveMasked(account, i) == 0)
=> getActiveUnmasked(account, i) != getBitmapCurrency(account)
```
TAUTOLOGICAL INVARIANT

If the bitmap currency is not zero, and the active currency is zero, then the bitmap and active currencies are different.

// BAD INVARIANT
assert 0 <= i && i < 9 &&
getBitmapCurrency(account) != 0 &&

(hasCurrencyMask(account, i) && getActiveUnmasked(account, i) == 0) ||
getActiveMasked(account, i) == 0)
=> getActiveUnmasked(account, i) != getBitmapCurrency(account)
TAUTOLOGICAL INVARIANT

Same tautological statement

// BAD INVARIANT
assert 0 <= i && i < 9 &&
getBitmapCurrency(account) != 0 &&
( // When a bitmap is enabled it can only have currency masks
  // in the active currencies bytes
  (hasCurrencyMask(account, i) && getActiveUnmasked (account, i) == 0) ||
  getActiveMasked(account, i) == 0 )
=> getActiveUnmasked(account, i) != getBitmapCurrency(account)
/// @notice Enables a bitmap currency for msg.sender, account cannot have any assets when this call
/// occurs. Will revert if the account already has a bitmap currency set.
/// @param currencyId the currency to enable the bitmap for.
/// @dev emit:AccountSettled emit:AccountContextUpdate
/// @dev auth:msg.sender

function enableBitmapCurrency(uint16 currencyId) external {
    require(msg.sender != address(this)); // dev: no internal call to enableBitmapCurrency
    require(currencyId <= maxCurrencyId); // dev: invalid currency id
    address account = msg.sender;
    (AccountContext memory accountContext, /* didSettle */) = _settleAccountIfRequired(account);
    accountContext.enableBitmapForAccount(currencyId, block.timestamp);
    accountContext.setAccountContext(account);
}
1. Enable a bitmap currency on your account, eg. ETH.

2. Deposit a second currency into your account, eg. DAI.

3. Call `enableBitmapForAccount` a second time, switching your bitmap currency to DAI. Due to a logic error, the system believes that it would have to check DAI twice in free collateral, effectively doubling the DAI collateral believed to be present in the account.

4. Borrow in significant amounts without sufficient collateral; drain funds
A non-zero asset **cannot** be both **bitmap** and **active**

// BAD INVARIANT
assert 0 <= i && i < 9 &&
getActiveUnmasked(account, i) != 0 &&
hasCurrencyMask(account, i)
=> getActiveUnmasked(account, i) != getBitmapCurrency(account)
1. The fixed invariant catches the bug in enableBitmapCurrency
2. The fixed invariant verifies the bug fix
3. The tautology automatic detection finds the problem in the bad invariant
TAKEAWAYS

Writing specifications is hard

Check your spec!
Human reviews
Automatic checks

Suspect, don’t trust

- When the prover reports a bug, it is always useful
- When you get a proof, be suspicious

A right specification can prevent Billion $ bugs
THANK YOU!

Uri Kirstein Software Engineer and Developer Relations