CERTORA

Move fast and break nothing
AUDITING AND FORMAL VERIFICATION

Better Together
TRUE/FALSE STATEMENTS

- Formal verification eliminates the need for auditing
- Auditing eliminates the need for formal verification
- Auditing can find bugs after formal verification
- Formal verification can find bugs after Audit
- Auditing cannot show the absence of bugs
- Formal Verification cannot show the absence of bugs
- Formal Verification cannot show the presence of bugs
AUDITING

- Review the documentation and the code
- Check against common mistakes
- Identify issues
- Evaluate the security of the process
SOMETHING COMPLETELY DIFFERENT: DIFF

```css
&.active .Team-list li{
    @include transition(all .5s);
    @include listTransitionDelay(100, 4, 200);
    @include transform(translateY(0));
    opacity: 1;
}
```

```css
&.active .Team-list li{
    @include transition(all .5s);
    @include listTransitionDelay(100, 8, 200);
    @include transform(translateY(0));
    opacity: 0.5;
}
```
FORMAL VERIFICATION ≈ SMART DIFF

- **CODE**
  - .SOL

- **SPEC**
  - .SPEC

**CERTORA PROVER**

- Proofs that every behavior meets the spec
- A rare behavior which violates the spec
The Dark Age of Formal Verification

The Golden Age of Formal Verification

25 Years of Hard Work

The Dark Age of Formal Verification

25 Years of Hard Work

The Golden Age of Formal Verification

1949

1970

1979

1989

2002

2006

2010

2011

2015

Program testing can show the presence of errors, not their absence

ROBIN

MILNER

TONY

HOARE

EDGAR

DIJKSTRA

PATRICK

COUSOT

ROBERT

W. FLOYD

Formal methods are the future of computer science. Always have been, always will be. William E. Aitken


Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability

BILL

GATES

VITALIK

BUTERIN

If a machine is expected to be infallible, it cannot also be intelligent.

ALAN

TURING

ROBIN

MILNER

TONY

HOARE

EDGAR

DIJKSTRA

PATRICK

COUSOT

ROBERT

W. FLOYD

RICHARD

A. DEMILLO

RICHARD

J. LIPTON

ALAN

J. PERLIS

If a machine is expected to be infallible, it cannot also be intelligent.

ALAN

TURING

ROBIN

MILNER

TONY

HOARE

EDGAR

DIJKSTRA

PATRICK

COUSOT

ROBERT

W. FLOYD

RICHARD

A. DEMILLO

RICHARD

J. LIPTON

ALAN

J. PERLIS

VITALIK

BUTERIN

Code is Law

WHY NOW
FORMAL SPECIFICATION / INVARIANTS

- Necessary parts of code documentation
- **Desired** properties of the code
- **What** the code is supposed to do
  - not how
Some examples
- No double spend
- Liquidation increases the amount of collateral
- The order of deposits does not matter

Useful for auditing

Specification Languages
- TLA+
- Assert and require(SMTChecker)
- Certora Verification Language(CVL)
INVARIANTS

Good Properties of the reachable states

\[ x + y \geq 2 \]
INTERESTING INVARIANTS DeFi

- For every borrowed token there is sufficient collateral borrow
- Sum of balances is equal to the total amount
- \( \text{LiquidityShares} > 0 \iff \text{SystemHolding} > 0 \)
- Each token has a unique entry (index in array)
- Reward does not exceed max value
BAD INVARIANTS

❖ Too permissive

\[ X + 2 > X \]
\[ 2 \cdot X < X \Rightarrow 5 > 7 \]

❖ Too restrictive

\[ X = 3 \lor X = 4 \]
See James’s lecture
GOOD PROPERTIES OF TRANSITIONS
REACHABLE STATES / HOARE TRIPLES

\( \{P\} \text{ Contract } \{Q\} \)

Every execution of the contract starting in a state in P results in a state in Q
TRANSACTIONAL INVARIANTS: GOOD PROPERTIES OF TRANSITIONS REACHABLE STATES / HOARE TRIPLES

\[
\{ P \} \text{ Contract } \{ Q \}
\]

Every execution of the contract starting in a state in P results in a state in Q.

```plaintext
if P then {
    Contract;
    assert Q;
}
```
SPEC VS. CODE

Declarative (what)
- Not meant to be executed
- Partial
- Reusable
- Capture the essence from a user perspective
- Readable and small
- Can have bugs

Imperative (how)
- Efficient
- Complete
- Tailor made
- Describes an efficient way to realize the specification
- Well documented but can be complex
- Can have bugs
**SIMPLE EXAMPLE**

**MONEY TRANSFER**

Code

```solidity
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:

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

Test

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

CORRECT MONEY TRANSFER

Code

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

Invariant:

\[\text{total} = \sum_a \text{balances}[a]\]

Proof

\[\sum a: \text{address} \text{old.balances}[a] = \sum a: \text{address} \text{new.balances}[a]\]
THINGS TO NOTE

- FV looks for behaviors which are **bad**
  - The developer specifies the **desired** behavior
  - The computer searches through **potentially infinite** behaviors

- FV is a hard (undecidable) computational problem

- But it can be automatically solved in many cases
  - Millions of variables
  - Developers can help with modularity
FORMAL VERIFICATION $\approx$ SMART DIFF

Proofs that every behavior meets the spec
Unknown (Timeout)
A rare behavior which violates the spec

- CODE
  - .SOL
- SPEC
  - .SPEC
MORE THINGS TO NOTE

- FV is the only method to prove properties
- FV is **useless** without good specs
- But sometimes generic rules suffice
WARNINGS WITH FV

- Some level of paranoia is useful
- When FV reports a bug check it
- When FV does not report a bug ⇒ Mutate the program and check
THE SUSHISWAP PROTOCOL

- Sophisticated protocol (established 2020)
- Decentralized exchange
- Tricky code & thousands of locs
- TVL $2.17B
KASHPAIR SIMPLIFIED

Inside the system

<table>
<thead>
<tr>
<th>User</th>
<th>User borrow</th>
<th>User collateral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 <=> 1.3

liquidate(Alice, Bob)
msg.sender = Bob

Alice’s loan is closed

<table>
<thead>
<tr>
<th>User</th>
<th>User borrow</th>
<th>User collateral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

External to the system

Bob has assets to cover the loan of Alice

System’s assets

Bob’s assets

Bob Borrow assets reduced and collateral asset increased

System receive Borrow for Collateral

Borrow token
Collateral token

Bob’s assets

Borrow token
Collateral token

System’s assets
KASHIPAIR SIMPLIFIED

```solidity
function batchCalls(address[] callee, bytes[] calldata datas) {
    callee[i].call(datas[i]);
}

/* Liquidation of a user that is in insolvent state. 
   user - address to liquidate 
   to - address to receive collateral */

function liquidate(address user, address to) {
    require (!_isSolvent(user));

    borrow = userBorrowAmount[user];
    collateral = userCollateralAmount[user];

    userBorrowAmount[user] = 0;
    userCollateralAmount[user] = 0;

    borrowToken.transferFrom(msg.sender, address(this), borrow);
    collateralToken.transfer(to, collateral);
}
```
GOOD PROPERTIES OF LIQUIDATION

- Collateral token balance and Borrow token balance are complementary

- If Borrow token balance increases, then Collateral token balance decreases

- Weaker requirements in practice
RULE: ANTIMONOTONICITY OF LIQUIDATION

\[
\{ \ b = \text{BorrowT.balanceOf(System)} \ \land \ c = \text{CollateralT.balanceOf(System)} \ \}
\]

\text{liquidate}(x, y)

\[
\{ \text{BorrowT.balanceOf(System)} > b \iff \text{CollateralT.balanceOf(System)} < c \}
\]

function liquidate(address user, address to) {
    require (!_isSolvent(user));

    borrow = userBorrowAmount[user];
    collateral = userCollateralAmount[user];

    userBorrowAmount[user] = 0;
    userCollateralAmount[user] = 0;

    borrowToken.transferFrom(msg.sender, address(this), borrow);
    collateralToken.transfer(to, collateral);
}

rule antimonotonicityOfLiquidation () {
    env e;
    address user;
    address to;

    collateralBefore = collateralToken.balanceOf(this);
    borrowBefore = borrowToken.balanceOf(this);
    ...
    liquidate(e, user, to);

    collateralAfter = collateralToken.balanceOf(this);
    borrowAfter = borrowToken.balanceOf(this);

    assert(borrowBefore < borrowAfter <=
        collateralBefore > collateralAfter);
}
CHECKING ANTIMONOTONOTICITY

### Results for SimpleBorrowSystem:

<table>
<thead>
<tr>
<th>Test name</th>
<th>Result</th>
<th>Time(secs)</th>
<th>Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>antimonotonicityOfLiquidation</td>
<td><img src="green" alt="Green" /></td>
<td>1</td>
<td><img src="green" alt="Green" /></td>
</tr>
<tr>
<td>envfreeFuncsAreNonpayable</td>
<td><img src="green" alt="Green" /></td>
<td>0</td>
<td><img src="green" alt="Green" /></td>
</tr>
<tr>
<td>envfreeFuncsStaticCheck</td>
<td><img src="green" alt="Green" /></td>
<td>0</td>
<td><img src="green" alt="Green" /></td>
</tr>
<tr>
<td>onlyCollateralCanBorrow</td>
<td><img src="green" alt="Green" /></td>
<td>0</td>
<td><img src="green" alt="Green" /></td>
</tr>
</tbody>
</table>

### Results for envfreeFuncsStaticCheck:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Result</th>
<th>Time(secs)</th>
<th>Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>userBorrowAmount(address)</td>
<td><img src="green" alt="Green" /></td>
<td>0</td>
<td><img src="green" alt="Green" /></td>
</tr>
<tr>
<td>userCollateralAmount(address)</td>
<td><img src="green" alt="Green" /></td>
<td>0</td>
<td><img src="green" alt="Green" /></td>
</tr>
</tbody>
</table>
Inside the system:

<table>
<thead>
<tr>
<th>User</th>
<th>User borrow</th>
<th>User collateral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 <=> 1.3

External to the system:

- Mallory does not need assets to cover the loan
- Mallory received assets

```
batchCalls([System],[liquidate(Mallory, Alice)]) msg.sender = Mallory

liquidate(Alice, Mallory) msg.sender = this
```

System “pays” itself for the loan

- System only gives Collateral
- System’s assets
- Bob’s assets

Alice has a loan

- User borrow
- User collateral

Alice’s loan is closed

- User borrow
- User collateral

- System’s assets
- Mallory’s assets
- Bob’s assets
function liquidate(address user, address to) {
    require (!_isSolvent(user));
    require (msg.sender != address(this));

    borrow = userBorrowAmount[user];
    collateral = userCollateralAmount[user];

    userBorrowAmount[user] = 0;
    userCollateralAmount[user] = 0;

    borrowToken.transferFrom(msg.sender, address(this), borrow);
    collateralToken.transfer(to, collateral);
}

rule antimonotonicityOfLiquidation () {
    env e;
    address user;
    address to;

    collateralBefore = collateralToken.balanceOf(this);
    borrowBefore = borrowToken.balanceOf(this);
    ... 

    liquidate(e, user, to);

    collateralAfter = collateralToken.balanceOf(this);
    borrowAfter = borrowToken.balanceOf(this);

    assert(borrowBefore < borrowAfter <=
        collateralBefore > collateralAfter);
}
# Checking Antimonicity of Fixed Code

## Results for SimpleBorrowSystem:

<table>
<thead>
<tr>
<th>Test name</th>
<th>Result</th>
<th>Time (secs)</th>
<th>Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>antimonoticityOfLiquidation</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
<tr>
<td>envfreeFuncsAreNonpayable</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
<tr>
<td>envfreeFuncsStaticCheck</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
<tr>
<td>onlyCollateralCanBorrow</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
</tbody>
</table>

## Results for envfreeFuncsStaticCheck:

<table>
<thead>
<tr>
<th>Function name</th>
<th>Result</th>
<th>Time (secs)</th>
<th>Dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>userBorrowAmount(address)</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
<tr>
<td>userCollateralAmount(address)</td>
<td>![绿勾]</td>
<td>0</td>
<td>![下载]</td>
</tr>
</tbody>
</table>
SushiSwap’s TRIDENT

Users interact with Pools via the Trident Router Contract:

Router Contract, connecting users and liquidity pools

Users

Liquidity Pools Contracts

>>> From a High-Level Invariant to a Pool-Draining Vulnerability in SushiSwap’s Trident <<<
SushiSwap’s TRIDENT INVARIANT

( totalLiquidity, tokenA.balanceOf(system), tokenB.balanceOf(system) = 0 )

∨

( totalLiquidity, tokenA.balanceOf(system), tokenB.balanceOf(system) ≠ 0 )

>> From a High-Level Invariant to a Pool-Draining Vulnerability in SushiSwap’s Trident <<
SushiSwap’s ALL BETS ARE OFF

- Token A
- Token B
- Liquidity tokens

External transfer
Burn single
Swap
A POTENTIAL SPEC OF AN OWNABLE SYSTEM

\{ \text{currentOwner} = \text{owner()} \land \text{other} \neq \text{currentOwner} \} \\
\text{Op} \\
\{ \text{owner()} \neq \text{currentOwner} \lor \text{owner()} \neq \text{other} \}
SUGGESTED SOFTWARE LIFE CYCLE

1. Design
2. Spec, Code, Test
3. Testing
4. Spec Checking
5. Fuzzing
6. Auditing
7. Formal Verification
**MYTHS vs REALITY ABOUT FORMAL VERIFICATION**

- ❌ FV can only prove absence of bugs
- ☑ Biggest value of FV is finding bugs
- ❌ Hardest problem is computational
- ☑ Hardest problem is specification
- ❌ FV is one-time deal
- ☑ FV guarantees code upgrade safety
TAKEAWAYS

- No silver bullet in code correctness
- FV and Auditing are required for code with high value
- Specification is the key
  - Code is law -> Spec is law
  - Simple invariants for complex code
TAKEAWAYS

- Tools improve security and time to market
  - Testing code
  - Testing specifications
  - Fuzzing
  - Formal verification of the code
  - Monitoring and simulating the specifications
THANK YOU