Zeus: Analyzing Safety of Smart Contracts

Sukrit Kalra, Seep Goel, Mohan Dhawan and Subodh Sharma

Tommy and Idan
Introduction

- Smart contracts are programs that run on the blockchain
- They are written in high-level languages such as Solidity
- Faithful execution of a smart contract is enforced by the blockchain’s consensus protocol
- Correctness and fairness of the smart contracts is not enforced by the blockchain, and should be verified by the developer
Correctness and Fairness

- Correctness means the code is accurate and complete, producing intended results without errors and bugs.
- Fairness means the code adheres to the agreed upon higher-level business logic for interaction. The code shouldn't be biased towards any party, and shouldn't allow any party to cheat.
Correctness and Fairness - Example

```solidity
while (Balance > (depositors[index].Amount * 115/100) && index<Total_Investors) {
    if(depositors[index].Amount!=0) {
        payment = depositors[index].Amount * 115/100;
        depositors[index].EtherAddress.send(payment);
        Balance -= payment;
        Total_Paid_Out += payment;
        depositors[index].Amount=0; // Remove investor
    }
    break;
}
```

The contract offers a 15% payout to any investor. Sadly, the contract has both fairness and correctness issues.
Correctness and Fairness - Example

while (Balance > (depositors[index].Amount * 115/100) && index<Total_Investors) {
    if(depositors[index].Amount!=0) {
        payment = depositors[index].Amount * 115/100;
        depositors[index].EtherAddress.send(payment);
        Balance -= payment; // --------------------------
        Total_Paid_Out += payment; // POTENTIAL OVERFLOW! 🚨🚨🚨
        depositors[index].Amount=0; // --------------------------
    }
    break;
}

Correctness issue: The contract has a potential overflow in the Total_Paid_Out variable.
Correctness and Fairness - Example

```java
while (Balance > (depositors[index].Amount * 115/100) && index<Total_Investors) {
    if(depositors[index].Amount!=0) {
        payment = depositors[index].Amount * 115/100;
        depositors[index].EtherAddress.send(payment);
        Balance -= payment;
        Total_Paid_Out += payment;
        depositors[index].Amount=0;
    } break;
}
```

Fairness issue (1): `index` is never incremented within the loop, and so the payout is made to just one investor.
Correctness and Fairness - Example

while (Balance > (depositors[index].Amount * 115/100) && index<Total_Investors) {
    if(depositors[index].Amount!=0)) {
        payment = depositors[index].Amount * 115/100;
        depositors[index].EtherAddress.send(payment);
        Balance -= payment;
        Total_Paid_Out += payment;
        depositors[index].Amount=0;
    } break; // <------------------------------------
}

Fairness issue (2): The break statement is inside the while statement, and so the loop will always break after the first iteration. Meaning, only the first investor will get paid. (Prob. the owner)
Incorrect Contracts - Reentrancy

```
contract Wallet {
    mapping(address => uint) private userBalances;
    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }
    // ...
}

contract AttackerContract {
    function () {
        Wallet wallet;
        wallet.withdrawBalance();
    }
}
```
Incorrect Contracts - Reentrancy

contract Wallet {
    mapping(address => uint) private userBalances;

    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            userBalances[msg.sender] = 0; // Mitigated by swapping the lines
            msg.sender.call(userBalances[msg.sender]);
        }
    }
    // ...
}

contract AttackerContract {
    function () {
        Wallet wallet;
        wallet.withdrawBalance();
    }
}
Incorrect Contracts - Unchecked Send

- Solidity allows only 2300 gas upon a send call
- Computation-heavy fallback function at the receiving contract will cause the invoking send to fail
- Contracts not handling failed send calls correctly may result in the loss of Ether
Incorrect Contracts - Unchecked Send

```java
if (gameHasEnded && !prizePaidOut) {
    winner.send(1000); // Send a prize to the winner
    prizePaidOut = True;
}
```

The `send` call may fail, but `prizePaidOut` is set to `True` regardless. Meaning the prize will never be paid out. 😞
Incorrect Contracts - Failed Send

- Best practices suggest executing a `throw` upon a failed `send`, in order to revert the transaction
- However, this may put contracts in risk
Incorrect Contracts - Failed Send

```
for (uint i=0; i < investors.length; i++) {
    if (investors[i].invested == min investment) {
        payout = investors[i].payout;
        if (!investors[i].address.send(payout)))
            throw;
        investors[i] = newInvestor;
    }
}
```

- A DAO that pays dividends to its smallest investor when a new investor offers more money, and the smallest is replaced
- A wallet with a fallback function that takes more than 2300 gas to run can invest enough to become the smallest investor
- No new investors will be able to join the DAO
Incorrect Contracts - Overflow/underflow

```solidity
uint payout = balance/participants.length;
for (var i = 0; i < participants.length; i++)
    participants[i].send(payout);
```

- `i` is of type `uint8`, and so it will overflow after 255 iterations
- Attacker can fill up the first 255 slots in the array, and gain payouts at the expense of other investors
Incorrect Contracts - Transaction State Dependence

- Contract writers can utilize transaction state variables, such as `tx.origin` and `tx.gasprice`, for managing control flow within a smart contract.
- `tx.gasprice` is fixed and is published upfront - cannot be exploited 😊
- `tx.origin` allows a contract to check the address that originally initiated the call chain.
Incorrect Contracts - Transaction State Dependence

```solidity
contract UserWallet {
    function transfer(address dest, uint amount) {
        if (tx.origin != owner)
            throw;
        dest.send(amount);
    }
}

contract AttackWallet {
    function() {
        UserWallet w = UserWallet(userWalletAddr);
        w.transfer(thief.Storage Addr, msg.sender.balance);
    }
}
```
Incorrect Contracts - Transaction State Dependence

contract UserWallet {
    function transfer(address dest, uint amount) {
        if (msg.sender != owner) // FIXED!
            throw;
        dest.send(amount);
    }
}

- **tx.origin** is the address of the original initiator of the call chain
- **msg.sender** is the address of the caller of the current function
Unfair Contracts - Absence of Logic

- Access to sensitive resources and APIs must be guarded, for instance:
  - `selfdestruct`:
    - Kill a contract and send its balance to a given address
    - Should be preceded by a check that only the owner of the contract is allowed to kill it
    - Several contracts did not have this check
Unfair Contracts - Incorrect Logic

```python
while (balance > persons[payoutCursor_Id_].deposit / 100 * 115) {
    payout = persons[payoutCursor_Id_].deposit / 100 * 115;
    persons[payoutCursor_Id].EtherAddress.send(payout);
    balance -= payout;
    payoutCursor_Id_ ++;
}
```

- Two similar variables, `payoutCursor_Id` and `payoutCursor_Id_`
- The deposits of all investors go to the 0th participant, possibly the person who created the contract
Unfair Contracts - Logically Correct but Unfair

Auction House Contract

```solidity
function placeBid(uint auctionId){
    Auction a = auctions[auctionId];
    if (a.currentBid >= msg.value)
        throw;
    uint bidIdx = a.bids.length++;
    Bid b = a.bids[bidIdx];
    b.bidder = msg.sender;
    b.amount = msg.value;
    // ...
    BidPlaced(auctionId, b.bidder, b.amount);
    return true;
}
```

- The contract does not disclose whether it is "with reserve" or not
- The seller can participate in the auction and artificially bid up the price
- The seller can withdraw the property from the auction before it is sold
ZEUS

- Takes as input a smart contract and a policy against which the smart contract must be verified
- Performs static analysis atop the smart contract code
- Inserts the policy predicates as asserts
- Converts the smart contract embedded with policy assertions to LLVM bitcode
- Invokes its verifier to determine assertion violations
Zeus Workflow
Formalizing Solidity Semantics

- Abstract language that captures relevant constructs of Solidity programs
- A program consists of a sequence of contract declarations.
- Each contract is abstractly viewed as a sequence of one or more method definitions
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @Id\{ \text{global } v : T; \text{ function } @Id(l : T) \{ S \}\}^* \]
\[ S ::= (l : T@Id)^* \mid l := e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x := \text{post function } @Id \ (l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestructure} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]

\[ C ::= \text{contract @Id}\{ \text{global } v : T; \text{ function@Id}(l : T) \{ S\}\}^* \]

\[ S ::= (l : T@Id)^* \mid l ::= e \mid S; S \]

\[ \mid \text{if } e \text{ then } S \text{ else } S \]

\[ \mid \text{goto } l \]

\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]

\[ \mid x ::= \text{post function@Id}(l : T) \]

\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]

- A program consists of a sequence of contract declarations
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @Id\{ \text{global } v : T; \text{ function}@Id(l : T) \{ S\}\}^* \]
\[ S ::= (l : T@Id)^* \mid l ::= e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x ::= \text{post function}@Id (l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]

- Each contract is abstractly viewed as a sequence of one or more method definitions
- Storage private to a contract, denoted by the keyword global
- Since T is generic, we lose no generality with a single variable
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } \text{@Id}\{ \text{global } v : T; \text{ function@Id}(l : T) \{ S\}\}* \]
\[ S ::= (l : T@Id)^* \mid l ::= e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x ::= \text{post function@Id } (l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @\text{Id}\{ \text{global } v : T ; \text{ function } @\text{Id}(l : T) \{ S \})^* \}
\[ S ::= (l : T@\text{Id})^* \mid l ::= e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x ::= \text{post function } @\text{Id} (l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestructor} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } \text{@Id}\{ \text{global } v : T; \text{ function}\text{@Id}(l : T) \{ S\}\}^* \]
\[ S ::= (l : T\text{@Id})^* \mid l ::= e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x ::= \text{post function}\text{@Id}(l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @Id \{ \text{global } v : T; \text{ function } @Id (l : T) \{ S \} \} \]
\[ S ::= (l : T@Id)^* \mid l := e \mid S; S \]
\[ \begin{align*}
\text{if } e \text{ then } S \text{ else } S \\
\text{goto } l \\
\text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \\
x := \text{post function}@Id (l : T) \\
\text{return } e \mid \text{throw} \mid \text{selfdestruct}
\end{align*} \]

- Regular if-then-else statements
An Abstract Language modeling Solidity

\[ P ::= C^* \]

\[ C ::= \text{contract \text{@Id}\{ global } v : T; \text{ function@Id}(l : T) \{ S\})^* \]

\[ S ::= (l : T\text{@Id})^* | l ::= e | S; S \]

| \text{if } e \text{ then } S \text{ else } S |
| \text{goto } l |
| \text{havoc } l : T | \text{assert } e | \text{assume } e |
| x ::= \text{post function@Id} (l : T) |
| \text{return } e | \text{throw} | \text{selfdestruct} |

• goto a given line
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract} @Id\{ \text{global} v : T; \text{function} @Id(l : T) \{ S\}\}^* \]
\[ S ::= (l : T@Id)^* \mid l := e \mid S; S \]
\[ \mid \text{if e then S else S} \]
\[ \mid \text{goto l} \]
\[ \mid \text{havoc l : T} \mid \text{assert e} \mid \text{assume e} \]
\[ \mid x := \text{post function} @Id (l : T) \]
\[ \mid \text{return e} \mid \text{throw} \mid \text{selfdestruct} \]

• Assigns a non-deterministic value
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @Id \{ \text{global } v : T; \text{ function } @Id (l : T) \{ S \} \}^* \]
\[ S ::= (l : T@Id)^* \mid l ::= e \mid S ; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x ::= \text{post function } @Id (l : T) \]
\[ \mid \text{return } e \mid \text{throw } \mid \text{selfdestruct} \]

• Check of truth value of predicates
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } \@\text{Id}\{ \text{global } v : T; \text{function } \@\text{Id}(l : T) \{ S \}\}^* \]
\[ S ::= (l : T@Id)^* \mid l := e \mid S; S \]
\[ \mid \text{if } e \text{ then } S \text{ else } S \]
\[ \mid \text{goto } l \]
\[ \mid \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ \mid x := \text{post function@Id}(l : T) \]
\[ \mid \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]

- Blocks until the supplied expression becomes true
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } @Id\{ \text{global } v : T; \text{function } @Id(l : T) \{ S \}\}^* \]
\[ S ::= (l : T@Id)^* | l ::= e | S; S \]
\[ \text{if } e \text{ then } S \text{ else } S \]
\[ \text{goto } l \]
\[ \text{havoc } l : T | \text{assert } e | \text{assume } e \]
\[ x ::= \text{post function } @Id (l : T) \]
\[ \text{return } e | \text{throw } | \text{selfdestruct} \]

- call() invocations (send with argument)
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract @Id}\{ \text{global } v : T; \text{ function}@Id(l : T) \{ S\}\}^* \]
\[ S ::= (l : T@Id)^* \mid l := e \mid S; S \]
\[ \text{if } e \text{ then } S \text{ else } S \]
\[ \text{goto } l \]
\[ \text{havoc } l : T \mid \text{assert } e \mid \text{assume } e \]
\[ x := \text{post function}@Id (l : T) \]
\[ \text{return } e \mid \text{throw} \mid \text{selfdestruct} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract}\ \text{@Id}\{ \text{global}\ v : T; \ \text{function}@Id(l : T)\ \{S\}\}\}^* \]
\[ S ::= (l :\ T@Id)^* \mid l := e \mid S; S \mid \text{if}\ e\ \text{then}\ S\ \text{else}\ S \mid \text{goto}\ l \mid \text{havoc}\ l : T \mid \text{assert}\ e \mid \text{assume}\ e \mid x := \text{post}\ \text{function}@Id\ (l : T) \mid \text{return}\ e \mid \text{throw}\ \mid \text{selfdestruct} \]
An Abstract Language modeling Solidity

\[ P ::= C^* \]
\[ C ::= \text{contract } \text{@Id}\{ \text{global } v : T; \text{ function}\text{@Id}(l : T) \{ S\})^* \]
\[ S ::= (l : T\text{@Id})^* | l := e | S; S \]
  \text{if } e \text{ then } S \text{ else } S
\text{goto } l
\text{havoc } l : T | \text{assert } e | \text{assume } e
\text{x := post function}\text{@Id}(l : T)
\text{return } e | \text{throw} | \text{selfdestruct}
Language Semantics

\(\langle \mathcal{T}, \sigma \rangle, \ BC \rangle\) - The blockchain state

- \(\langle \mathcal{T}, \sigma \rangle\) - The block \(B\) being currently mined
- \(\mathcal{T}\) - The completed transactions that are not committed
- \(\sigma\) - The global state of the system after executing \(\mathcal{T}\)
- \(BC\) - The list of committed blocks

\(\sigma : id \rightarrow g, g \in Vals\)

- \(id\) - Identifier of the contract
- \(g\) - Valuation of global variable
Language Semantics

$\gamma$ - A transaction defined as a stack of frames $f$

\[ f ::= \langle \ell, id, M, pc, v \rangle \] - A frame

- $\ell \in Vals$ - The valuation of the method local variables $l$
- $M$ - The code of the contract with identifier $id$
- $pc$ - The program counter
- $v : \langle i, o \rangle$ - Auxiliary memory for storing input and output
Language Semantics

- \( c := (\gamma, \sigma) \) - The configuration, captures the state of the transaction
- \( \rightsquigarrow \) - Small step operation
- \( \rightarrow \) - Transaction relation for globals and blockchain state
- \( \leftarrow \) - Assignment
Language Semantics

\[
\begin{align*}
\text{Post-Invoke} & :& \text{LookupStmt}(M, pc) = \text{post } \text{fn}c@\text{Id}^\prime(i^\prime), \\
& & f = \langle \ell, \text{Id}, M, pc, \langle i, * \rangle \rangle, \ c = \langle f.A, \sigma \rangle \ \\
& & f^\prime \leftarrow \langle \ell', \text{Id}', M', pc', \langle i', * \rangle \rangle \\
& & c \rightsquigarrow c[i \mapsto f'.f.A]
\end{align*}
\]

\[
\begin{align*}
\text{Post-Return-Succ} & :& \text{LookupStmt}(M', pc') = \text{return } e, \\
& & f' = \langle \ell', \text{Id'}, M', pc', \langle i', 1 \rangle \rangle, \ c = \langle f'.f.A, \sigma \rangle \\
& & f \leftarrow \langle \ell, \text{Id}, M, pc, \langle i, * \rangle \rangle \\
& & c \rightsquigarrow c[i \mapsto f[pc \mapsto pc + 1, \ell \mapsto \ell_{\text{new}}].A]
\end{align*}
\]

\[
\begin{align*}
\text{Post-Return-Fail} & :& \text{LookupStmt}(M', pc') = \text{throw}, \\
& & f' = \langle \ell', \text{Id'}, M', pc', \langle i', 0 \rangle \rangle, \ c = \langle f'.f.A, \sigma \rangle \\
& & f \leftarrow \langle \ell, \text{Id}, M, pc, \langle i, * \rangle \rangle \\
& & c \rightsquigarrow c[f[pc \mapsto pc + 1, \ell \mapsto \ell_{\text{new}}].A]
\end{align*}
\]

\[
\begin{align*}
\text{Self-dest} & :& \text{LookupStmt}(M', pc') = \text{selfdestruct} \\
& & f' \leftarrow \langle \ell', \text{Id'}, M', pc', \langle i', * \rangle \rangle, \ c = \langle f'.f.A, \sigma \rangle \\
& & \text{del } \text{Id'}, c \rightsquigarrow c[f[pc \mapsto pc + 1].A]
\end{align*}
\]

\[
\begin{align*}
\text{Assert} :& & \text{LookupStmt}(M, pc) = \text{assert } e \\
& & f \leftarrow \langle \ell, \text{Id}, M, pc, \langle i, * \rangle \rangle, \ c = \langle f.e, \sigma \rangle \\
& & c \rightsquigarrow c[f[pc \mapsto pc + 1].A] \\
& & \langle y, \sigma \rangle \rightsquigarrow^* \langle \epsilon, \sigma' \rangle, \\
& & T \leftarrow \gamma \\
& & B \vdash B[T \mapsto T \cup \{T\}, \sigma \mapsto \sigma']
\end{align*}
\]

\[
\begin{align*}
\text{Tx-Success} :& & \text{lookupStmt}(M, pc) = \text{throw}, \\
& & f \leftarrow \langle \ell, \text{Id}, M, pc, \langle i, \perp \rangle \rangle, \ c = \langle f.e, \sigma \rangle \\
& & c \rightsquigarrow c[f[pc \mapsto \epsilon]]
\end{align*}
\]

\[
\begin{align*}
\text{Add-block} :& & \langle \langle T', \sigma \rangle, BC \rangle, \langle \epsilon, \sigma \rangle \\
& & \langle \langle T', \sigma \rangle, BC \rangle \rightarrow \langle \langle \epsilon, \sigma \rangle, BC.T \rangle
\end{align*}
\]
Policy Example

```xml
<Subject> msg.sender </Subject>
<Object> a.seller </Object>
<Operation trigger="pre"> placeBid </Operation>
<Condition> a.seller != msg.sender </Condition>
<Result> True </Result>
```

```solidity
function placeBid(uint auctionId){
    Auction a = auctions[auctionId];
    if (a.currentBid >= msg.value)
        throw;
    uint bidIdx = a.bids.length++;
    Bid b = a.bids[bidIdx];
    b.bidder = msg.sender;
    b.amount = msg.value;
    // ...
    BidPlaced(auctionId, b.bidder, b.amount);
    return true;
}
```
Formalizing the Policy Language

- $PVars$ - The set of program variables
- $Func$ - The set of function names in a contract
- $Expr$ - The set of conditional expressions
Formalizing the Policy Language

- **Policy specification**: \( \langle \text{Sub}, \text{Obj}, \text{Op}, \text{Cond}, \text{Res}, \rangle \)
  - \( \text{Sub} \in PV\text{ar} \) - The set of source variables (one or more) that need to be tracked
  - \( \text{Obj} \in PV\text{ar} \)
  - \( \text{Op} := \langle f, \text{trig} \rangle, f \in \text{Func}, \text{trig} \in \{\text{pre, post}\} \)
  - \( \text{Cond} \in Expr \)
  - \( \text{Res} \in \{T, F\} \)
Formalizing the Policy Language

• **Policy specification:** \( \langle Sub, Obj, Op, Cond, Res, \rangle \)
  
  ○ \( Sub \in PVar \)
  
  ○ \( Obj \in PVar \) - The set of variables representing entities with which the subject interacts
  
  ○ \( Op := \langle f, \text{trig} \rangle, f \in Func, \text{trig} \in \{pre, post\} \)
  
  ○ \( Cond \in Expr \)
  
  ○ \( Res \in \{T, F\} \)
Formalizing the Policy Language

- **Policy specification**: \( \langle Sub, Obj, Op, Cond, Res, \rangle \)
  - \( Sub \in PVar \)
  - \( Obj \in PVar \)
  - \( Op := \langle f, trig \rangle, f \in Func, trig \in \{pre, post\} \) - The set of side-affecting invocations that capture the effects of interaction between the subject and the object
  - \( Cond \in Expr \)
  - \( Res \in \{T, F\} \)
Formalizing the Policy Language

• **Policy specification:** \( \langle Sub, Obj, Op, Cond, Res, \rangle \)

  ○ \( Sub \in PVar \)
  ○ \( Obj \in PVar \)
  ○ \( Op := \langle f, trig \rangle, f \in Func, trig \in \{pre, post\} \)
  ○ \( Cond \in Expr \) - The set of predicates that govern this interaction leading to the operation
  ○ \( Res \in \{T, F\} \)
Formalizing the Policy Language

- **Policy specification**: $(Sub, Obj, Op, Cond, Res, )$
  - $Sub \in PV a r$
  - $Obj \in PV a r$
  - $Op := \langle f, trig\rangle, f \in Func, trig \in \{pre, post\}$
  - $Cond \in Expr$
  - $Res \in \{T, F\}$ - Indicates whether the interaction between the subject and operation as governed by the predicates is permitted or constitutes a violation
## Translation To LLVM

<table>
<thead>
<tr>
<th>AST Node</th>
<th>Abstract</th>
<th>LLVM API</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContractDefinition</td>
<td>contract@Id{...}</td>
<td>Module</td>
</tr>
<tr>
<td>EventDefinition</td>
<td>function@Id(T){S}</td>
<td>FunctionType, Function</td>
</tr>
<tr>
<td>FunctionDefinition</td>
<td>function@Id(T){S}</td>
<td>FunctionType, Function</td>
</tr>
<tr>
<td>Block</td>
<td>{S}</td>
<td>BasicBlock</td>
</tr>
<tr>
<td>VariableDeclarationStatement</td>
<td>(I:T)*</td>
<td>CreateStore, CreateExtOrTrunc</td>
</tr>
<tr>
<td>VariableDeclaration</td>
<td>(I:T)</td>
<td>GlobalVariable, CreateAlioca</td>
</tr>
<tr>
<td>Literal</td>
<td>t</td>
<td>ConstantInt</td>
</tr>
<tr>
<td>Return</td>
<td>return e</td>
<td>ReturnInst, CreateExtOrTrunc, CreateGEP</td>
</tr>
<tr>
<td>Assignment</td>
<td>l := e</td>
<td>CreateExtractValue, CreateExtOrTrunc, CreateLoad, CreateStore, CreateBinOp</td>
</tr>
<tr>
<td>ExpressionStatement</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>Id</td>
<td>ValueSymbolTable, GlobalVariable, getFunction</td>
</tr>
<tr>
<td>IfStatement</td>
<td>if e then S else S</td>
<td>BasicBlock, CreateBr, CreateCondBr</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>goto or post</td>
<td>CreateExtOrTrunc, CreateCall, Function</td>
</tr>
<tr>
<td>WhileStatement / ForStatement</td>
<td>if e then goto l else S</td>
<td>BasicBlock, CreateCondBr</td>
</tr>
<tr>
<td>StructDefintion</td>
<td>T</td>
<td>StructType</td>
</tr>
<tr>
<td>Throw</td>
<td>throw</td>
<td>Function, CreateCall</td>
</tr>
<tr>
<td>Break / Continue</td>
<td>if e then goto l</td>
<td>CreateBr</td>
</tr>
</tbody>
</table>
Implementation

- The Policy builder: 500 lines of code
- The translator from solidity to LLVM: 3000 lines of code
- The code was written on C++ using the Abstract Syntax Tree (AST) derived from the Solidity compiler solc
- Verifier: Verifiers that are already work with LLVM like SMACK, Seahorn
**End-to-End Example**

function transfer() {
    msg.sender.send(msg.value);
    balance = balance - msg.value;
}

<havoc value>
<havoc balance>
B@δ() {
    assert(value <= balance)
    post B'@δ()
    balance = balance - value
}
define void @transfer() {
  entry:
  % value = getelementptr %msgRecord* @msg, i32 0, i32 4
  %0 = load i256* % value
  %1 = load i256* @balance
  %2 = icmp ule i256 %0, %1
  br i1 %2, label %"75", label %"74"
"74":
  call void @VERIFIER error()
  br label %"75"
"75":
  % sender = getelementptr %msgRecord* @msg, i32 0, i32 2
  %3 = load i160* % sender
  %4 = call i1 @send(i160 %3, i256 %0)
  %5 = sub i256 %1, %0
  store i256 %5, i256* @balance
  ret void
}
define void @main() {
  entry:
  %0 = call i256 @_VERIFIER_NONDET ()
  store 1256 %0, 1256* @balance
  //...
}
define void @transfer() {
  entry:
    % value = getelementptr %msgRecord* @msg, i32 0, i32 4
    %0 = load i256* % value // Load msg.value into %0
    %1 = load i256* @balance // Load balance into %1
    %2 = icmp ule i256 %0, %1 // Compare %0 and %1 (%2 = 1 if %0 <= %1)
    br i1 %2, label "%75", label "%74" // Branch based on %2
"74": // An assert failure is modeled as a call to the verifier’s error function
  call void @ VERIFIER error()
function
  br label "%75"
"75": // If %2 is 1 (i.e., value <= balance)
  % sender = getelementptr %msgRecord* @msg, i32 0, i32 2
  %3 = load i160* % sender
  %4 = call i1 @send(i160 %3, i256 %0) // Call send
  %5 = sub i256 %1, %0 // balance -= value
  store i256 %5, i256* @balance // Store updated balance
  ret void
}
define void @main() {
  entry: // Globals are automatically havoc-ed to explore the entire data domain
    %0 = call i256 @_VERIFIER_NONDET ( )
    store 1256 %0, 1256* @balance
    // ...
}
Handling Correctness Bugs
contract Wallet {
    mapping(address => uint) private userBalances;
    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }
}

contract AttackerContract {
    function () {
        Wallet wallet;
        wallet.withdrawBalance();
    }
}
Handling Correctness Bugs - Reentrancy

contract Wallet {
    mapping(address => uint) private userBalances;
    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }
    // ...
}
Handling Correctness Bugs - Reentrancy

contract Wallet {
    mapping(address => uint) private userBalances;

    function withdrawBalance2() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            assert(false);
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }

    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            withdrawBalance2();
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }
}
contract Wallet {
    mapping(address => uint) private userBalances;
    function withdrawBalance2() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            assert(false); // Now it's unreachable
            msg.sender.call(userBalances[msg.sender]);
            userBalances[msg.sender] = 0;
        }
    }
    function withdrawBalance() {
        uint amountToWithdraw = userBalances[msg.sender];
        if (amountToWithdraw > 0) {
            userBalances[msg.sender] = 0; // The safe version :)
            withdrawBalance2();
            msg.sender.call(userBalances[msg.sender]);
        }
    }
}
Handling Correctness Bugs - Unchecked Send

// Globals ...
prizePaidOut = False;

if (gameHasEnded && !prizePaidOut) {
    winner.send(1000); // May fail, thus the Ether is lost forever :(
    prizePaidOut = True;
}
Handling Correctness Bugs - Unchecked Send

// Globals ...
prizePaidOut = False;
checkSend = True;

if (gameHasEnded && !prizePaidOut) {
    checkSend &= winner.send(1000); // False if send fails
    assert(checkSend);
    prizePaidOut = True;
}
Initializing a global variable `checkSend` to `true`

Take logical AND of `checkSend` and the result of each `send`

For every write of a global variable, assert that `checkSend` is `true`
// Globals ...
investors = [ ... ];

for (uint i=0; i < investors.length; i++) {
    if (investors[i].invested == min investment) {
        payout = investors[i].payout;
        if (!(investors[i].address.send(payout)))
            throw;
        investors[i] = newInvestor;
    }
}
Handling Correctness Bugs - Failed Send

// Globals ...
investors = [ ... ];
checkSend = True;

for (uint i=0; i < investors.length; i++) {
    if (investors[i].invested == min investment) {
        payout = investors[i].payout;
        if (!(checkSend &= investors[i].address.send(payout)))
            assert(checkSend);
            throw;
        investors[i] = newInvestor;
    }
}

• Same as unchecked send, but assert that checkSend is true before throw's
• Indicates a possibility of reverting the transaction due to control flow reaching a throw on a failed send
Limitations

- Fairness properties involving mathematical formulae are harder to check
  - ZEUS depends on the user to give appropriate policy
- Zeus is not faithful exactly to Solidity syntax
  - Does not explicitly account for runtime EVM parameters such as gas
  - `throw` and `selfdestruct` are modeled as program exit
- Zeus does not analyze contracts with an assembly block
  - Only 45 out of 22,493 contracts in the data set use it
- Zeus does not support virtual functions in contract hierarchy (i.e. `super`)
  - Only 23 out of 22,493 contracts in the data set use it
# Evaluation

<table>
<thead>
<tr>
<th>Bug</th>
<th>Safe</th>
<th>Unsafe</th>
<th>No Result</th>
<th>Timeout</th>
<th>False +ve</th>
<th>False -ve</th>
<th>% False Alarms</th>
<th>Safe</th>
<th>Unsafe</th>
<th>No Result</th>
<th>Timeout</th>
<th>False +ve</th>
<th>False -ve</th>
<th>% False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrancy</td>
<td>1438</td>
<td>54</td>
<td>7</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>548</td>
<td>265</td>
<td>226</td>
<td>485</td>
<td>254</td>
<td>51</td>
<td>31.24</td>
</tr>
<tr>
<td>Unc. snd</td>
<td>1191</td>
<td>324</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0.20</td>
<td>1066</td>
<td>112</td>
<td>203</td>
<td>143</td>
<td>89</td>
<td>188</td>
<td>7.56</td>
</tr>
<tr>
<td>Failed send</td>
<td>1068</td>
<td>447</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int. overflow</td>
<td>378</td>
<td>1095</td>
<td>18</td>
<td>33</td>
<td>40</td>
<td>0</td>
<td>2.72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx. State Dep.</td>
<td>1513</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blk. State Dep.</td>
<td>1266</td>
<td>250</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
<td>798</td>
<td>15</td>
<td>226</td>
<td>485</td>
<td>2</td>
<td>84</td>
<td>0.25</td>
</tr>
<tr>
<td>Tx. Order Dep.</td>
<td>894</td>
<td>607</td>
<td>13</td>
<td>10</td>
<td>16</td>
<td>0</td>
<td>1.07</td>
<td>668</td>
<td>129</td>
<td>222</td>
<td>485</td>
<td>116</td>
<td>158</td>
<td>14.20</td>
</tr>
</tbody>
</table>
Zeus's Performance

![CDF graphs showing performance over time and constraints]

- Zeus
- Oyente

Time (min)

CDF

# constraints

Reentrancy
Unchecked send
Failed send
Integer overflow
Tx. state dep.
Blk. state dep.
Tx. order dep
Conclusion

- 94.6% of 22.4K contracts are vulnerable
- ZEUS is sound (zero false negative)
- Low false positive rate
- ZEUS is fast (less than 1 min to verify 97% of the contracts)
Thank you for listening! ⚡ ⚡