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

```
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

```
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

```
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

```
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

```
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(thiefStorageAddr, 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

```
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

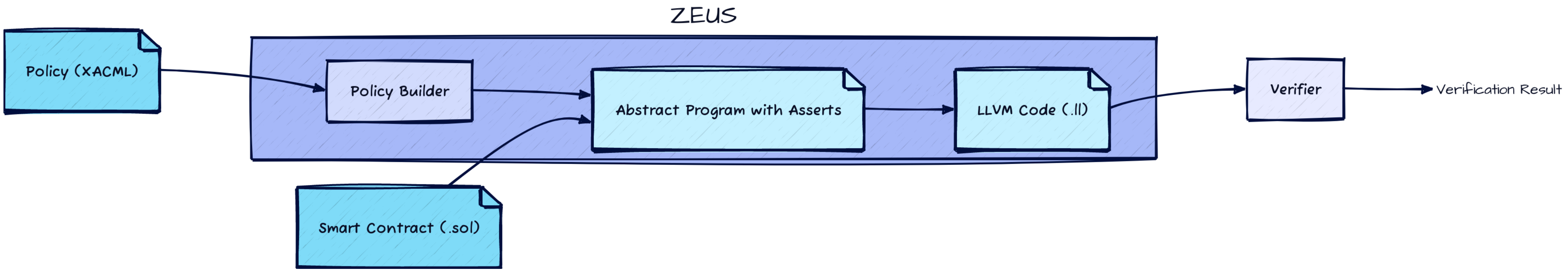
```
function placeBid(uint auctionId){
    Auction a = auctions[auctionId];
    if (a.currentBid >= msg.value)
        throw;
    uint bidIdx = a.bids.length++;
    Bid b = a.bids[bidIdx];
    b.bidder = msg.sender;
    b.amount = msg.value;
    // ...
    BidPlaced(auctionId, b.bidder, b.amount);
 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

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

An Abstract Language modeling Solidity

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

- A program consists of a sequence of contract declarations

An Abstract Language modeling Solidity

$$P ::= C^*$$

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

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

$$\mid \textbf{if } e \textbf{ then } S \textit{ else } S$$

$$\mid \textbf{goto } l$$

$$\mid \textbf{havoc } l : T \mid \textbf{assert } e \mid \textbf{assume } e$$

$$\mid x := \textbf{post function@Id } (l : T)$$

$$\mid \textbf{return } e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

- 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

$$P ::= C^*$$

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

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

$$\mid \textbf{if } e \textbf{ then } S \textit{ else } S$$

$$\mid \textbf{goto } l$$

$$\mid \textbf{havoc } l : T \mid \textbf{assert } e \mid \textbf{assume } e$$

$$\mid x := \textbf{post function@Id } (l : T)$$

$$\mid \textbf{return } e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

An Abstract Language modeling Solidity

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

An Abstract Language modeling Solidity

$$P ::= C^*$$

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

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

$$\mid \textbf{if } e \textbf{ then } S \textit{ else } S$$

$$\mid \textbf{goto } l$$

$$\mid \textbf{havoc } l : T \mid \textbf{assert } e \mid \textbf{assume } e$$

$$\mid x := \textbf{post function@Id } (l : T)$$

$$\mid \textbf{return } e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

An Abstract Language modeling Solidity

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

- Regular if-then-else statements

An Abstract Language modeling Solidity

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

- goto a given line

An Abstract Language modeling Solidity

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

- Assigns a non-deterministic value

An Abstract Language modeling Solidity

$$P ::= C^*$$

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

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

$$\mid \textbf{if}\ e\ \textbf{then}\ S\ else\ S$$

$$\mid \textbf{goto}\ l$$

$$\mid \textbf{havoc}\ l\ :\ T \mid \textbf{assert}\ e \mid \textbf{assume}\ e$$

$$\mid x := \textbf{post function@Id}\ (l\ :\ T)$$

$$\mid \textbf{return}\ e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

- Check of truth value of predicates

An Abstract Language modeling Solidity

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

- Blocks until the supplied expression becomes true

An Abstract Language modeling Solidity

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

- call() invocations (send with argument)

An Abstract Language modeling Solidity

$$P ::= C^*$$

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

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

$$\mid \textbf{if}\ e\ \textbf{then}\ S\ else\ S$$

$$\mid \textbf{goto}\ l$$

$$\mid \textbf{havoc}\ l : T \mid \textbf{assert}\ e \mid \textbf{assume}\ e$$

$$\mid x := \textbf{post function@Id}\ (l : T)$$

$$\mid \textbf{return}\ e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

An Abstract Language modeling Solidity

$$P ::= C^*$$

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

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

$$\mid \textbf{if } e \textbf{ then } S \textit{ else } S$$

$$\mid \textbf{goto } l$$

$$\mid \textbf{havoc } l : T \mid \textbf{assert } e \mid \textbf{assume } e$$

$$\mid x := \textbf{post function@Id } (l : T)$$

$$\mid \textbf{return } e \mid \textbf{throw} \mid \textbf{selfdestruct}$$

An Abstract Language modeling Solidity

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

Language Semantics

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

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

$\sigma : id \rightarrow g\ ,\ g \in Vals$

- id - Identifier of the contract
- g - Valuation of global variable

Language Semantics

γ - 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

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

Language Semantics

Post-Invoke

$$\frac{\begin{array}{c} LookupStmt(M, pc) = (x := \textbf{post}\ fnc@Id'(i')), \\ f = \langle \ell, Id, M, pc, \langle i, * \rangle\rangle,\ \ c = \langle f.A, \sigma\rangle \\ f' \leftarrow \langle \ell', Id', M', 0, \langle i', * \rangle\rangle \end{array}}{c \rightsquigarrow c[\gamma \mapsto f'.f.A]}$$

Post-Return-Succ

$$\frac{\begin{array}{c} LookupStmt(M', pc') = \textbf{return}\ \ \textbf{e}, \\ f' = \langle \ell', Id', M', pc', \langle i', 1\rangle\rangle,\ \ c = \langle f'.f.A, \sigma\rangle \\ f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle \end{array}}{c \rightsquigarrow c[\gamma \mapsto f[pc \mapsto pc + 1, \ell \mapsto \ell_{new}].A]}$$

Post-Return-Fail

$$\frac{\begin{array}{c} LookupStmt(M', pc') = \ \textbf{throw}, \\ f' \leftarrow \langle \ell', Id', M', pc', \langle i', 0\rangle\rangle,\ \ c = \langle f'.f.A, \sigma\rangle \\ f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle \end{array}}{c \rightsquigarrow c[f[pc \mapsto pc + 1, \ell \mapsto \ell_{new}].A]}$$

Self-destruct

$$\frac{\begin{array}{c} LookupStmt(M', pc') = \textbf{selfdestruct} \\ f' \leftarrow \langle \ell', Id', M', pc', \langle i', * \rangle\rangle,\ \ c = \langle f'.f.A, \sigma\rangle \end{array}}{del\ \ Id', c \rightsquigarrow c[f[pc \mapsto pc + 1].A]}$$

Assert

$$\frac{\begin{array}{c} LookupStmt(M, pc) = \textbf{assert}\ \ \textbf{e} \\ f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle,\ \ c = \langle f.A, \sigma\rangle \end{array}}{c \rightsquigarrow c[f[pc \mapsto pc + 1].A]}$$

Tx-Success

$$\frac{\begin{array}{c} \langle \gamma, \sigma\rangle \rightsquigarrow^{*} \langle \epsilon, \sigma'\rangle, \\ T \leftarrow \gamma \end{array}}{B \rightarrow B[\mathcal{T} \mapsto \mathcal{T} \cup \{T\}, \sigma \mapsto \sigma']}$$

Tx-Failure

$$\frac{\begin{array}{c} LookupStmt(M, pc) = \ \textbf{throw}, \\ f \leftarrow \langle \ell, Id, M, pc, \langle i, \bot\rangle\rangle,\ \ c = \langle f.\epsilon, \sigma\rangle \end{array}}{c \rightsquigarrow c[f.\epsilon \mapsto \epsilon]}$$

Add-block

$$\frac{\langle\langle \mathcal{T}, \sigma\rangle, BC\rangle, \langle \epsilon, \sigma\rangle}{\langle\langle \mathcal{T}, \sigma\rangle, BC\rangle \rightarrow \langle\langle \epsilon, \sigma\rangle, BC.\mathcal{T}\rangle}$$

Policy Example

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

```
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**: $\langle Sub, Obj, Op, Cond, Res, \rangle$
  - $Sub \in PVar$ - The set of source variables (one or more) that need to be tracked
  - $Obj \in PVar$
  - $Op := \langle f, trig \rangle, f \in Func, trig \in \{pre, post\}$
  - $Cond \in Expr$
  - $Res \in \{T, F\}$

Formalizing the Policy Language

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

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**: $\langle Sub, Obj, Op, Cond, Res, \rangle$
  - $Sub \in PVar$
  - $Obj \in PVar$
  - $Op := \langle f, trig \rangle, f \in Func, trig \in \{pre, post\}$
  - $Cond \in Expr$
  - $Res \in \{T, F\}$ - Indicates whether the interaction between the subject and operation as governed by the predicates is permitted or constitutes a violation

Translation To LLVM

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

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;
}
```

```
<Subject> msg.value </Subject>
<Object> msg.sender </Object>
<Operation trigger="pre"> send </Operation>
<Condition> msg.value <= balance </Condition>
<Result> True </Result>
```

```
havoc value
havoc balance
B@δ() {
    assert(value <= balance)
    post B'@δ()
    balance = balance - value
}
```

End-to-End Example

```
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
    //...
}
```

End-to-End Example

```
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

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;
        }
    }
    // ...
}
```

```
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;
        }
    }
}
```

Handling Correctness Bugs - Reentrancy

```
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;
}
```

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;
}
```

- Initialize 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`

Handling Correctness Bugs - Failed Send

```
// 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`

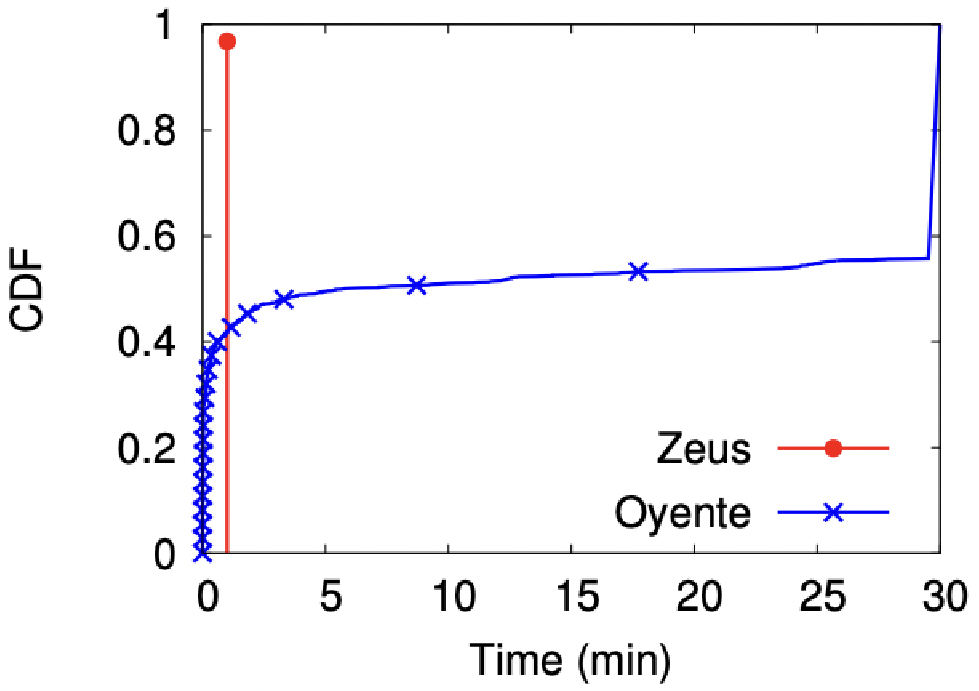
Limitations

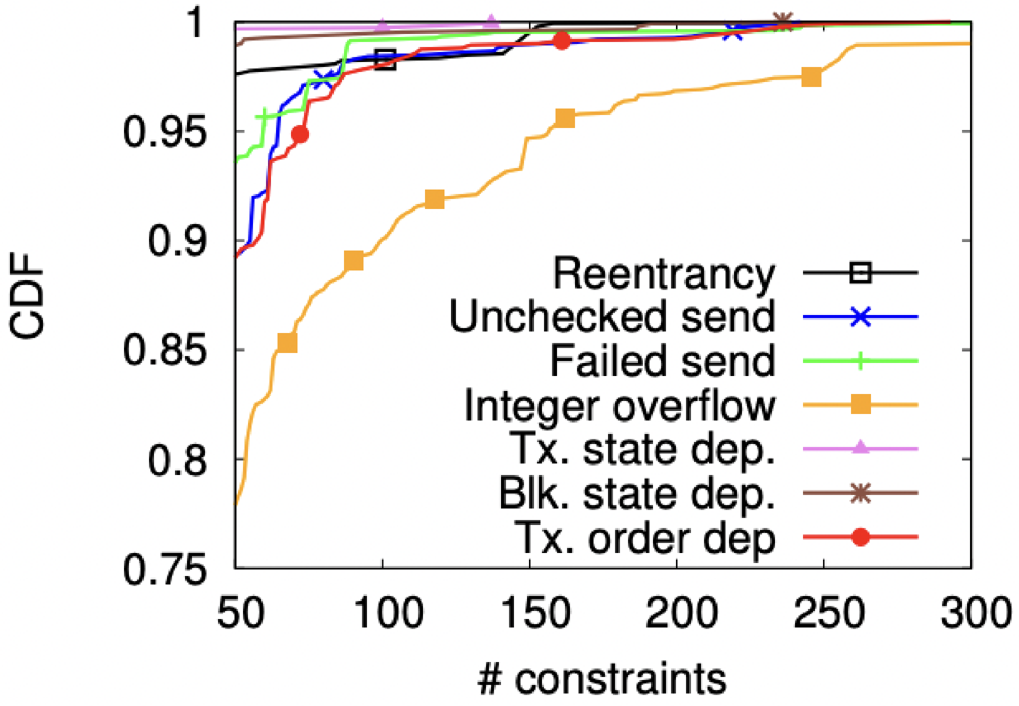
- Fairness properties involving mathematical formulae are harder to check
  - ZEUS depends on the user to give appropriate policy
- Zeus is not faithful exactly to Solidity syntax
  - Does not explicitly account for runtime EVM parameters such as gas
  - `throw` and `selfdestruct` are modeled as program 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

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

Zeus's Performance





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! ⚡ ⚡