ZEUS: Analyzing Safety of Smart Contracts

Sukrit Kalra
IBM Research
sukrit.kalra@in.ibm.com

Seep Goel
IBM Research
sgoel219@in.ibm.com

Mohan Dhawan
IBM Research
mohan.dhawan@in.ibm.com

Subodh Sharma
IIT Delhi
svs@iitd.ac.in

***Abstract*—A smart contract is hard to patch for bugs once it is deployed, irrespective of the money it holds. A recent bug caused losses worth around $50 million of cryptocurrency. We present ZEUS—a framework to verify the correctness and validate the fairness of smart contracts. We consider correctness as adherence to safe programming practices, while fairness is adherence to agreed upon higher-level business logic. ZEUS leverages both abstract interpretation and symbolic model checking, along with the power of constrained horn clauses to quickly verify contracts for safety. We have built a prototype of ZEUS for Ethereum and Fabric blockchain platforms, and evaluated it with over 22.4K smart contracts. Our evaluation indicates that about 94.6% of contracts (containing cryptocurrency worth more than $0.5 billion) are vulnerable. ZEUS is sound with zero false negatives and has a low false positive rate, with an order of magnitude improvement in analysis time as compared to prior art.**

I. INTRODUCTION

Blockchain is the design pattern that underpins the Bitcoin cryptocurrency [70]. However, its use of consensus to validate interaction amongst participant nodes is a key enabler for applications that require mutually distrusting peers to conduct business without the need for a trusted intermediary. One such use is to enable a smart contract, which programatically encodes rules to reflect any kind of multi-party interaction. With over $1.4 billion invested in blockchain last year [3], and the increasing trend towards autonomous applications, smart contracts are fast becoming the preferred mechanism to implement financial instruments (e.g., currencies, derivatives, wallets, etc.) and applications such as decentralized gambling.

While the faithful execution of a smart contract is enforced by the blockchain's consensus protocol, it remains the prerogative of the participating entities to (i) verify the smart contract's *correctness*, i.e., the syntactic implementation follows the best practices, and (ii) validate its *fairness*, i.e., the code adheres to the agreed upon higher-level business logic for interaction. While manual auditing of contracts for correctness is possible to an extent, it still remains laborious and error prone. Automatic formal auditing, on the other hand, requires specialized tools and logic. The problem is exacerbated by the fact that smart contracts, unlike other distributed systems code, are immutable and hard to patch in case of bugs, irrespective of the money they hold. For example, investors in TheDAO [43] lost cryptocurrency worth around $50 million because of a bug in the code that allowed an attacker to repeatedly siphon off money [44]. In this paper, we tackle the problem of formal verification of smart contracts, since reasoning about their correctness and fairness is critical before their deployment.

```
(1)  while (Balance > (depositors[index].Amount * 115/100)
                          && index<Total_Investors) {
(2)    if(depositors[index].Amount!=0)) {
(3)      payment = depositors[index].Amount * 115/100;
(4)      depositors[index].EtherAddress.send(payment);
(5)      Balance -= payment;
(6)      Total_Paid_Out += payment;
(7)      depositors[index].Amount=0; //remove investor
(8)    } break;
(9)  }
```

Fig. 1: An unfair contract (adapted from [31]).

The smart contract in Fig. 1 advertises a 15% profit payout to any investor. However, the contract has both correctness and fairness issues. First, the arithmetic operation in line 6 can potentially overflow, which is a correctness bug. Second, the variable `index` never increments within the loop, and thus the payout is made to just one investor. Finally, the `break` statement exits the loop after payment to the first investor, who is the contract owner. Thus, the contract does not payback any other investor. The last two bugs result in fairness issues.

Most prior art in the area of smart contracts deals with security and/or privacy concerns in designing them [62], [64], [67], [73]. There is, however, little work that analyzes smart contracts for vulnerabilities [50], [54], [68]. Oyente [68] uses symbolic execution for bug detection at the bytecode level, but it is neither sound nor complete. Thus, it can result in several false alarms even in trivial contracts, as we observed and communicated to Oyente's developers [26]. Since it is very hard to recreate the intent from the bytecode alone (due to loss of contextual information such as types, reuse of same bytecode for different function calls, etc.) several fairness and correctness issues, including integer overflow/underflow amongst others, are thus completely ignored by Oyente. Further, it conservatively handles loops [1] [29], [32] (with a bound of one) resulting in under approximation of loop behavior, and thus fails to detect the two fairness bugs in Fig. 1.

Bhargavan *et al.* [54] propose a framework to formally verify smart contracts written in a subset of Solidity using F*, which leaves out important constructs, such as loops. Considering the 22,493 contracts that we analyzed, around 93% contained loops. Thus, their tool will operate on a fraction of publicly available contracts, which is also corroborated by

[1] Oyente is under active development and future releases could add more features and reduce the false alarms.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23082
www.ndss-symposium.org

their results; they could evaluate only 46 out of 396 contracts. While use of F* may enable reasoning about most correctness and fairness properties, the authors suggest that such reasoning may require manual proofs. Although it is unclear when such situations may arise. In contrast, we establish that completely automated verification enables analysis of published contracts at a much larger scale. Why3 [50] is an experimental tool for formal verification of Solidity contracts, which is under active development and supports only a small subset of the entire syntax [22]. Further, Solidity to Why3 translation has not yet been tested and thus cannot be trusted [21].

We present the design and implementation of ZEUS—a practical framework for automatic formal verification of smart contracts using abstract interpretation and symbolic model checking. ZEUS takes as input the smart contracts written in high-level languages and leverages user assistance to help generate the correctness and/or fairness criteria in a XACML-styled template [51]. It translates these contracts and the policy specification into a low-level intermediate representation (IR), such as LLVM bitcode [66], encoding the execution semantics to correctly reason about the contract behavior. It then performs static analysis atop the IR to determine the points at which the verification predicates (as specified in the policy) must be asserted. Finally, ZEUS feeds the modified IR to a verification engine that leverages constrained horn clauses (CHCs) [55], [61], [69] to quickly ascertain the safety of the smart contract.

ZEUS leverages three key observations to be both sound and scalable. First, while the blockchain has execution akin to a concurrent system with task-based semantics, a transaction comprises of just one call chain starting from a publicly visible function in the smart contract. This observation helps significantly reduce the state space exploration for verifying most properties. Also, data dependence across transactions, such as read/write hazards among persistent state variables, requires analyzing $O(n^2)$ pairs of transaction interleavings. Second, smart contracts are both control- and data-driven. Thus, modeling contracts using abstract interpretation along with symbolic model checking allows ZEUS to soundly reason about program behavior. Abstract interpretation computes loop and function summaries over data domains, which are then used during the model checking phase that now operates upon a reduced state space. Lastly, CHCs provide a suitable mechanism to represent verification conditions, which can be discharged efficiently by SMT solvers.

ZEUS also benefits greatly from verification atop LLVM bitcode. Not only does this allow ZEUS to leverage an industry strength tool-chain for analysis, it enables ZEUS to plug in any verifier that operates upon the standardized (and formally verified [74]) LLVM bitcode. Use of LLVM bitcode also helps ZEUS to support verification of smart contracts for different blockchain platforms, including Ethereum [13] and Hyperledger Fabric [24] (or Fabric), written in diverse high-level languages, such as C#, GO and JAVA. Note that most high-level languages have mature source code to LLVM bitcode translators already available. We leverage LLVM's rich API set to develop the first Solidity to LLVM bitcode translator, which faithfully implements execution semantics for majority of the Solidity syntax for verification. Furthermore, use of LLVM passes allow ZEUS to separate translation from implementation of verification checks.

This paper makes the following contributions:

(**1**) We classify several new and previously known issues but unstudied in the context of smart contracts and show that they can potentially lead to loss of money (§ III).
(**2**) We present a formal abstraction of Solidity's execution semantics for verifying smart contracts using a combination of abstract interpretation and symbolic model checking (§ IV).
(**3**) We present the design and implementation of ZEUS (§ IV, and § V), a symbolic model checking framework for verification of correctness and fairness policies. We build the first Solidity to LLVM bitcode translator and provide a program analysis module that automatically inserts verification conditions given a policy specification. We also provide abstraction strategies to correctly model Solidity's execution semantics to ensure soundness. Further, we build an interactive predicate extraction tool to make it easy to specify policies for multi-party interactions in smart contracts.
(**4**) We present the first large scale source code analysis of Solidity-based smart contracts. Our evaluation (§ VI) with 22,493 Solidity smart contracts (of which 1524 were unique) indicates that about 94.6% of them (with a net worth of over $0.5 billion) are vulnerable to one or more correctness issues. However, we do not investigate the practical exploitability of these bugs. Additionally, we selected several representative contracts and applied contract-specific fairness criteria.
(**5**) ZEUS is sound (with zero false negatives) and outperforms Oyente for contracts in our data set, with low false positive rate and an order of magnitude improvement in time for analysis.
(**6**) We show ZEUS's generic applicability by leveraging it to verify smart contracts for the Fabric blockchain. We also demonstrate the ease of applying ZEUS to a verifier of choice by using SMACK [72] for verification (§ VI-D).

II. BACKGROUND

Blockchain is a distributed, shared ledger that records transactions between multiple, often mutually distrusting parties in a verifiable and permanent way. These transactions are maintained in a continuously growing list of ordered "blocks", which are tamper-proof and support non-repudiation. Apart from the array of transactions, each block also includes state metadata, including the creation timestamp, the Merkle hash of the transactions, the hash of the previous block in the chain, and smart contract code and data. Mining is the process of distributed, computational review performed on each block, enabling consensus in a mutually distrusting environment.

PERMISSIONLESS BLOCKCHAIN. Permissionless platforms allow anyone to join the network and participate in the process of block verification to create consensus. Examples of permissionless blockchain platforms include Bitcoin and Ethereum, where any miner can join the network and start mining. Permissionless platforms use consensus mechanisms, such as proof-of-work or proof-of-stake, to build trust and validate transactions. Since the permissionless blockchain is decentralized, anonymous and equally accessible to anyone, the ability to create trust and the ability to scale is low, resulting in low network throughput.

PERMISSIONED BLOCKCHAIN. A permissioned blockchain platform, such as Fabric, restricts the participants who can contribute to the consensus of the system state. In other

```
(1)  contract Wallet {
(2)    mapping(address => uint) private userBalances;
(3)    function withdrawBalance() {
(4)      uint amountToWithdraw = userBalances[msg.sender];
(5)      if (amountToWithdraw > 0) {
(6)        msg.sender.call(userBalances[msg.sender]);
(7)        userBalances[msg.sender] = 0;
(8)      }
(9)    }
(9)    ...
(10) }
```

```
(1)  contract AttackerContract {
(2)    function () {
(3)      Wallet wallet;
(4)      wallet.withdrawBalance();
(5)    }
(6)  }
```

Fig. 2: Same-function reentrancy attack.

words, only a restricted set of approved participants have the right to validate transactions. This restricted model provides better privacy, scalability and fine grained access control over users and their data. Hence, most private blockchains for financial institutions and other enterprises follow this model. Permissioned blockchains do not typically use proof-based mining to reach a consensus since all the actors are known; instead they use consensus algorithms such as RAFT [71], Paxos [65] or PBFT [57] to achieve higher network throughput.

III. Motivation

We describe the broad classes of correctness and fairness issues in smart contracts. We also describe potential attacks due to correctness bugs that can be exploited to gain financial benefits. None of the attacks discussed exploit any blockchain or Solidity vulnerabilities or compiler implementation bugs.

A. Incorrect Contracts

An incorrect contract uses constructs or programming paradigms that are not well understood in the context of the blockchain platform, resulting in a loss of money.

(I) Reentrancy. A function is reentrant if it can be interrupted while in the midst of its execution, and safely re-invoked even before its previous invocations complete execution. However, Solidity does not support concurrency, nor are there any interrupts that can halt a function execution. In spite of these safeguards, Solidity allows multiple parallel external invocations, which can invoke the same function using the `call` family of constructs, i.e., `call`, `callcode` and `delegatecall` [2]. If an externally invokable function does not correctly manage the global state, it will be susceptible to a same function reentrancy attack, such as TheDAO bug [1]. Reentrancy attacks can also happen if a contract's global state is not correctly managed across invocations of two different functions that operate upon the same global state. This bug is called cross-function race condition [8].

While both `call` and `send` can be used for transfer of Ether [3], `send` cannot cause reentrancy because `send` limits the fallback function to 2300 gas, which neither allows any

[2]Without loss of generality, we use `call` to refer to these constructs.

[3]Ether is Ethereum's virtual currency. Gas is the execution fee for every operation made on Ethereum.

```
(1)  if(gameHasEnded && !prizePaidOut) {
(2)    winner.send(1000); // send a prize to the winner
(3)    prizePaidOut = True;
(4)  }
```

Fig. 3: Unchecked `send` [46].

```
(1)  for (uint i=0; i < investors.length; i++) {
(2)    if (investors[i].invested == min_investment) {
(2)      payout = investors[i].payout;
(3)      if (!(investors[i].address.send(payout)))
(4)        throw;
(5)      investors[i] = newInvestor;
(6)    }
(7)  }
```

Fig. 4: Failed `send` [38].

storage write nor function calls [6], [15]. Oyente [68], however, considers the `CALL` bytecode to trigger its check for reentrancy. Since, both `send` and `call` map to the same `CALL` bytecode, Oyente generates several false alarms for reentrancy.

Attack. Fig. 2 shows a snippet of this vulnerability. The attacker invokes the fallback function [4] transferring control to the `Wallet`'s `withdrawBalance` function that uses the `call` construct to send Ether to the caller, thereby invoking the attacker's fallback function again. This repeated invocation siphons off Ether from the `wallet`'s balance. This attack can be mitigated by swapping lines 6 and 7.

(II) Unchecked send. Since Solidity allows only 2300 gas upon a `send` call, a computation-heavy fallback function at the receiving contract will cause the invoking `send` to fail. Contracts that do not correctly handle such failed invocations and allow modifications to the global state following the failed `send` call, may incorrectly lead to loss of Ether [46].

Attack. Consider the example in Fig. 3. The `send` method can fail, in which case the winner does not get the money, but `prizePaidOut` is set to `True`. Thus, the condition in line 1 is always `False` and the real winner can never claim the prize.

(III) Failed send. Best practices [16] suggest executing a `throw` upon a failed `send`, in order to revert the transaction. However, this paradigm can also put contracts at risk.

Attack. Consider Fig. 4, which describes a DAO that has a certain number of investors, and is at full capacity. If a new investor comes along and offers more money than the current smallest investor, the DAO will pay the dividend to the smallest, and put the new one in, increasing its capital stake. However, an adversarial wallet with a fallback function that takes more than 2300 gas to run, can lock this function by merely investing enough to become the smallest investor. When the adversarial wallet is next due to be booted off, the contract will fail while returning the money and `throw`, reverting all changes. This causes the wallet to still be a part of the investors, thereby causing loss of money to the DAO.

(IV) Integer overflow/underflow. Smart contracts primarily operate upon arithmetic operations, such as iterating over an array or computing balance amounts to send to a participant. However, since Solidity is strongly typed, implicit extending of signed or unsigned integers (e.g., from 8 byte `int` to 16 byte `int`) to store the result is not allowed,

[4]An anonymous function which is invoked if no matching method is found.

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

Fig. 5: Integer overflow [7].

```
(1)  contract UserWallet {
(2)    function transfer(address dest, uint amount) {
(3)      if (tx.origin != owner) { throw; }
(4)      dest.send(amount);
(5)    }
(6)  }
```

```
(1)  contract AttackWallet {
(2)    function() {
(3)      UserWallet w = UserWallet(userWalletAddr);
(4)      w.transfer(thiefStorageAddr, msg.sender.balance);
(5)    }
(6)  }
```

Fig. 6: tx.origin bug [45].

```
(1)  contract Wallet {
(2)    uint256 balance;
       ... // initialize balance
(3)    function checkAndPay(bytes32 sol,
                            address dest, uint amt) {
(4)      balance -= amt;
(5)      if (<solution != correct>) { throw; }
(6)      dest.send(amt);
(7)    }
(8)  }
```

Fig. 7: Unchecked resources.

```
(1)  while (balance >
           persons[payoutCursor_Id_].deposit/100*115) {
(2)    payout = persons[payoutCursor_Id_].deposit/100*115;
(3)    persons[payoutCursor_Id].EtherAddress.send(payout);
(4)    balance -= payout;
(5)    payoutCursor_Id_++;
(6)  }
```

Fig. 8: Variable mixup [47].

thereby causing all arithmetic operations to be susceptible to overflow/underflow. There are over 20 different scenarios that require careful handling of integer operations [20].

ATTACK. Fig. 5 highlights the severity of the problem. Specifically, the type of `i` will be `uint8`, because this is the smallest type that is available to hold the value 0. If there are more than 255 participants, then at `i=255`, `i++` will wrap around and return to 0. This will cause the payout to be sent to only the first 255 participants. An attacker can fill up these spots and gain payouts at the expense of other investors.

(V) 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. Since `tx.gasprice` is fixed and is published upfront to the miner, it cannot be exploited for profit. However, use of `tx.origin` for detecting the contract caller can make the contract vulnerable. For example, if we have a chain of calls, `msg.sender` points to the caller of the last function in the call chain. Solidity's `tx.origin` attribute allows a contract to check the address that originally initiated the call chain, and not just the last function call [35], [45].

ATTACK. Fig. 6 lists a snippet of code highlighting the bug. `UserWallet` is the contract that a user uses to dispense money, while the attacker deploys the `AttackWallet` contract. The attack requires the user to invoke the `AttackWallet`, which is possible with some social engineering or phishing techniques. When the `AttackWallet` instantiates `UserWallet` and invokes `transfer`, the `tx.origin` check at line 3 fails, since the originator of the call chain is the owner. If `tx.origin` were replaced by `msg.sender`, the check would succeed, and prevent the malicious contract from siphoning off money.

B. Unfair Contracts

We found several examples of syntactically correct contracts that do not implement the desired logic. Additionally, we also found examples of logically correct contracts that are unfair due to the subtleties involved in multi-party interaction.

(I) ABSENCE OF LOGIC. Access to sensitive resources and APIs must be guarded. For example, the `selfdestruct` is a sensitive call that is used to kill a contract and send its balance to a designated address. Thus, this call should be preceded by a check that only the owner of the contract is allowed to kill it. However, we observed that several contracts that used `selfdestruct` did not have this check, potentially allowing an adversary to receive money and kill the contract.

Consider another example as shown in Fig. 7. The contract `Wallet` defines a function `checkAndPay` that takes in a solution to a puzzle, a destination address, and an amount to send to that address, if the solution is correct. It also decrements the balance from the owner's account. If the balance in the wallet is less than the amount to be sent, then the owner gets the solution to his puzzle and not pay anything because `send` will fail. Thus, from the perspective of the solution provider, this contract is unfair. The problem can be easily remedied if there were appropriate checks before every write to a shared resource. For example, the contract writer can check if the the balance is less than the amount before line 4 and `throw`, thereby reverting the entire transaction and not accessing the solution. In general, contract writers can adhere to the following 3 step rule: check prerequisites, update state variables, and perform actions.

(II) INCORRECT LOGIC. There are many syntactically legal ways to achieve semantically unfair behavior. While there are several real-world examples for this class, in the interest of space we briefly describe four representative bugs.

- Consider the example in Fig. 8. Notice that two similar variables, `payoutCursor_Id_` and `payoutCursor_Id` are initialized to 0. The first one gets incremented, but the payouts go to the second one (see line 2), which stays at zero. Hence, the contract is not actually fair: the deposits of all investors go to the 0^{th} participant, possibly the person who created the scheme, and everyone else gets nothing.
- HackersGold, another popular contract, recently had a bug discovered [23] where the `transferFrom` function did not correctly increment the balance to be transferred to the recipient. The bug involved a typographical error where += was coded as =+, resulting in no increment in balance to be sent to the receiver. We found 15 unique contracts in our data set that include the same `transferFrom` functionality and hold over $\$35,000$ worth of Ether with over 6500 transactions executed between them. We continue to see several transactions even months after the issue was advertised.

```
(1)  if (balances[msg.sender] < value &&
                    value < 1208925819614629174706176) {
(2)    balances[msg.sender] -= value;
(3)    balances[to] = value;
(4)  }
```

```
(1)  if (balances[msg.sender] >= value &&
                    value < 1208925819614629174706176) {
(2)    balances[msg.sender] -= value;
(3)    balances[to] = value;
(4)  }
```

Fig. 9: Logic error in contracts [4] and [28].

```
(1)   function placeBid(uint auctionId)
                      returns (bool success) {
(2)     Auction a = auctions[auctionId];
(3)     if (a.currentBid >= msg.value) throw;
(4)     uint bidIdx = a.bids.length++;
(5)     Bid b = a.bids[bidIdx];
(6)     b.bidder = msg.sender;
(7)     b.amount = msg.value;
(8)     ...
(9)     BidPlaced(auctionId, b.bidder, b.amount);
(10)    return true;
(11)  }
```

Fig. 10: An unfair auction house contract (adapted from [2]).

• A recent attack [34] on the popular `MultiSig` wallet contract allowed an attacker to change the owner of the wallet by invoking the `initWallet` function in the context of the previous owner, which did not check for double initialization and was inadvertently made a public function. The attackers were able to get away with over $30mn worth of Ether.
• Consider Fig. 9 that shows the exact same snippet in two contracts [4] and [28]. However, the check to determine that `balance` must be greater than `value` to allow the transfer is incorrect in the first one and correct in the other one.

(IV) LOGICALLY CORRECT BUT UNFAIR. Consider the `placeBid` function from an auction house contract [2] in Fig. 10. By law, an auction in the U.S can be "with reserve" or "without reserve". If a seller is allowed to bid, the auction is "with reserve", which can affect the participants' willingness (since the seller can artificially bid up the price). Further, the seller may withdraw the property from the auction anytime prior to it being sold. However, most importantly, at such "with reserve" auctions, the seller may bid only if that right is disclosed to the participants. This contract does not disclose whether it is "with reserve" or not, and the knowledge is gleaned only by analyzing the source code. The `placeBid` function places no restriction on bidders willing to bid, indicating that sellers can also participate. A careful code analysis reveals that the seller can indeed withdraw the item before being sold. However, unsuspecting bidders, having no expertise in analyzing code, may lose money due to artificially increased bids or forfeit their participation fee. This contract is thus unfair to participants, and indicates the subtleties involved in multi-party interactions, where fairness is subjective.

C. Miner's Influence

A miner in a permissionless blockchain can order the transactions from his pool. A malicious miner can re-order transactions (while being adversarial to some participants) and obtain profit by prioritizing his own transactions.

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

Fig. 11: An abstract language modeling Solidity.

(I) BLOCK STATE DEPENDENCE. Solidity defines several block state variables, such as `timestamp`, `coinbase`, `number`, `difficulty` and `gaslimit`, which can be used to generate randomness [48]. All these variables are determined from the block header, and are thus, in principle, vulnerable to tampering by the block miner, who can insert suitable values to favor payouts intended for him, albeit with varying degrees of success [14]. While prior work [68] considers only `timestamp`, other block state variables can also lead to different Ether flows along different program paths.

(II) TRANSACTION ORDER DEPENDENCE. Concurrent systems have for long grappled with the problem of data races due to transaction ordering. While Solidity does not support concurrency, a miner can influence the outcome of a transaction due to its own reordering criteria. Since this dependence on transaction ordering is a universal blockchain feature, we consider it a limitation rather than a bug.

IV. ZEUS

ZEUS's tool chain for smart contract verification consists of (a) policy builder, (b) source code translator, and (c) verifier. Specifically, ZEUS takes as input a smart contract and a policy (written in a specification language) against which the smart contract must be verified. It performs static analysis atop the smart contract code and inserts the policy predicates as `assert` statements at correct program points. ZEUS then leverages its source code translator to faithfully convert the smart contract embedded with policy assertions to LLVM bitcode. Finally, ZEUS invokes its verifier to determine assertion violations, which are indicative of policy violations.

We now present a formal overview of ZEUS's workflow and present proofs of its soundness. While we focus on Solidity-based smart contracts, ZEUS's design is generic and applicable to contracts written in any source language. Note that our formalism is inspired from [68] to maintain readability.

A. Formalizing Solidity Semantics

We define an abstract language that captures relevant constructs of Solidity programs (see Fig. 11). A program consists of a sequence of contract declarations. Each contract is abstractly viewed as a sequence of one or more method definitions in addition to declaration and initialization of persistent storage private to a contract, denoted by the keyword `global`. A contract is uniquely identified by *Id*, where *Id* belongs to a set of *identifiers*. This invocation of the contract's publicly visible methods is viewed as a transaction.

For simplicity, we have methods with a single input variable of type T (where $Dom(T) \subseteq \mathbb{N}$) and a single variable

that is global to the contract's functions [5]. Since T is generic, it can represent collections and `structs` as well. Method invocations in Solidity can be of three types: internal, external and `call`. Internal and external invocations are modeled via the `goto` instruction or are inlined, while the `call` invocation is modeled separately as `post`. The body of a contract method is inductively defined by S. In contrast, the `post` statement can be invoked with arguments across contracts. Hence, argument l (of type T) is part of `post`.

The semantics of our language abstract concrete values and operations. Thus, enumeration of T or particular expression language e remains unspecified for us. Note that the details of the expression language are not important; one can assume linear arithmetic expressions defined for any traditional imperative language. The statement `havoc` assigns a non-deterministic value to the variable l. An `assert` statement introduces a check of truth value of predicates in the symbolic encoding. An `assume` statement blocks until the supplied expression becomes true and specifies a data state at a given control location in a contract.

While a formal argument about the semantic equivalence of Solidity and our abstract language is desirable and can be established by defining abstraction functions from Solidity constructs to constructs in our abstract language, we omit it in the interest of space. Instead, we intuitively reason about the various Solidity constructs and their equivalent modeling in our abstract language. Constructs such as `class`, `library` and `interfaces` can be desugared as a collection of global variables and functions in our abstract language. Even compilers model them similarly when translating C/C++ code to LLVM bitcode. `struct`, `mapping`, arrays and bytes are mapped to globals. Built-in methods such as `sha256` that affect the state of the same contract are modeled as external functions. Functions that operate upon addresses such as `send`, `transfer`, and `call` family of instructions are modeled via the `post` statement. Special constructs like `selfdestruct` are natively modeled in our abstract language. All control structures including function modifiers in Solidity can be desugared into `if-then-else` and `goto`. Solidity-style exception handling using `assert`, `require` and `throw` also maps directly to our abstract language using `if-then` and `throw`. Note that `assert` and `assume` in our language are used for verification, and `assert` has semantics different from those in Solidity. Other compiler directives such as `constant` and `storage` are also desugared, and are thus not modeled explicitly in our abstract language.

LANGUAGE SEMANTICS. The blockchain state is defined by the tuple: $\langle\langle\mathcal{T},\sigma\rangle, BC\rangle$ where $\langle\mathcal{T},\sigma\rangle$ is the block B being currently mined. BC is the list of *committed* blocks, and $\mathcal{T}$ denotes the multiset of *completed* transactions that are not yet committed. Let $Vals \subseteq \mathbb{N}$ be the set of values that expressions can take after evaluations. σ is the global state denoted by the function $\sigma : Id \rightarrow g$ that maps contract identifiers to a valuation of the global variables, where $g \in Vals$. Note that σ is the state of the system reached after executing $\mathcal{T}$ in an order as specified by the miner. Finally, each miner will add B to their respective copies of blockchain once it is validated.

A transaction is defined as a stack of frames represented by γ. Each frame is further defined as: $f := \langle \ell, id, M, pc, \nu\rangle$, where $\ell \in Vals$ is the valuation of the method-local variables l, M is the code of the contract with the identifier id, pc is the program counter, and $\nu := \langle i, o\rangle$ is an auxiliary memory for storing input and output. The top frame of γ is the frame under active execution and is relevant to the currently executing transaction; it is not part of the persistent blockchain state. An empty frame is denoted by ϵ. A configuration c, defined as $c := \langle\gamma,\sigma\rangle$, captures the state of transaction execution and $\rightsquigarrow$ denotes the small-step operational semantics.

Table 1 lists *relevant* semantic rules that govern changes in the configuration. Rules for remaining sequential statements are standard. The symbol $\rightarrow$ is overloaded to illustrate a transition relation for globals and blockchain states. The symbol $\leftarrow$ indicates an assignment to an `lvar`.

B. Formalizing the Policy Language

Assume $PVars$ to be the set of program variables, $Func$ to be a set of function names in a contract (which is uniquely identified by Id as defined in § IV-A) and $Expr$ to be the set of conditional expressions specified as quantifier-free first order logic (FOL) formulae. Policy specification must use these syntactic symbols to avoid any ambiguity during verification.

ZEUS leverages user assistance to build a XACML-styled five tuple policy specification [51] consisting of $\langle Sub, Obj, Op, Cond, Res\rangle$. Subject $Sub \in PVars$ is the set of source variables (one or more) that need to be tracked. Object $Obj \in PVars$ is the set of variables representing entities with which the subject interacts. Operation Op is the set of side-affecting invocations that capture the effects of interaction between the subject and the object. Op also specifies a trigger attribute, either 'pre' or 'post', indicating whether the predicates should hold before or after the specified operation. In other words, $Op := \langle f, trig\rangle$ where $f \in Func$ and $trig \in \{pre, post\}$. Condition $Cond \in Expr$ is the set of predicates that govern this interaction leading to the operation. Finally, $Res \in \{T, F\}$ indicates whether the interaction between the subject and operation as governed by the predicates is permitted or constitutes a violation.

TRANSLATION OF POLICY TO ASSERTIONS. Our abstract language includes assertions for defining state reachability properties on the smart contract. ZEUS leverages the policy tuple to extract: (a) predicate (i.e., $Cond$) to be asserted, and (b) the correct control location for inserting the `assert` statements in the program source.

Notice that $Cond$ is an expression in our abstract language. Thus, taking this expression predicate and wrapping it under `assert(exp)` creates a statement in our abstract language. Res indicates whether the condition will appear in its normal or negated form in the `assert` statement. Op indicates the function invocations where the predicates (as indicated by the condition) must satisfy $trig$. This $trig$ attribute along with Sub and Obj precisely discriminates which invocations of the operation should be prefixed or suffixed with the condition. In other words, we precisely know the control locations in the abstract program P where $Cond$ must be asserted.

More formally, $f : Sub \times Obj \times Op \rightarrow Loc$ where Loc is the set of program locations. Operationally, this function is

[5] We lose no generality with single local and global variables

Rule	Semantics	Description
Post-Invoke	$\frac{LookupStmt(M, pc) = (x := \textbf{post}\ fnc@Id'(i')),\ f = \langle \ell, Id, M, pc, \langle i, * \rangle\rangle,\ c = \langle f.A, \sigma\rangle \quad f' \leftarrow \langle \ell', Id', M', 0, \langle i', * \rangle\rangle}{c \rightsquigarrow c[\gamma \mapsto f'.f.A]}$	This rule creates a new frame f' and adds it to the top of the stack of frames $f.A$. Calling a function does not lead to any change in the global state σ as illustrated by the sequent. Note that frames can only be created when a client invokes a contract's publicly visible methods or via the post instruction.
Post-Return-Succ	$\frac{LookupStmt(M', pc') = \textbf{return e},\ f' = \langle \ell', Id', M', pc', \langle i', 1\rangle,\ c = \langle f'.f.A, \sigma\rangle \quad f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle}{c \rightsquigarrow c[\gamma \mapsto f[pc \mapsto pc + 1, \ell \mapsto \ell_{new}].A]}$	The rule updates the output of f' to 1 and pops it from the stack of frames, updates the value of local variable (from ℓ to ℓ_{new}) in the calling frame f, and finally updates pc of the procedure corresponding to f.
Post-Return-Fail	$\frac{LookupStmt(M', pc') = \textbf{throw},\ f' \leftarrow \langle \ell', Id', M', pc', \langle i', 0\rangle\rangle,\ c = \langle f'.f.A, \sigma\rangle \quad f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle}{c \rightsquigarrow c[f[pc \mapsto pc + 1, \ell \mapsto \ell_{new}].A]}$	The procedure for f' throws an exception. This results in the output of f' being updated to 0. In the calling frame f, the pc is advanced by 1 and local variable valuation is updated with the output of f'. The frame f' itself is removed from the stack frame.
Self-destruct	$\frac{LookupStmt(M', pc') = \textbf{selfdestruct}\ f' \leftarrow \langle \ell', Id', M', pc', \langle i', * \rangle\rangle,\ c = \langle f'.f.A, \sigma\rangle}{del\ Id', c \rightsquigarrow c[f[pc \mapsto pc + 1].A]}$	When a method in contract with identifier Id' issues a selfdestruct, it causes the associated frame f' to immediately pop off. The pc of the calling frame is advanced and the contract Id' is registered for deletion.
Assert	$\frac{LookupStmt(M, pc) = \textbf{assert e}\ f \leftarrow \langle \ell, Id, M, pc, \langle i, * \rangle\rangle,\ c = \langle f.A, \sigma\rangle}{c \rightsquigarrow c[f[pc \mapsto pc + 1].A]}$	Assert instruction only leads in advancement of the pc. The real use for assert is in generating verification conditions for the method.
Tx-Success	$\frac{\langle \gamma, \sigma\rangle \rightsquigarrow^* \langle \epsilon, \sigma'\rangle,\ T \leftarrow \gamma}{B \rightarrow B[\mathcal{T} \mapsto \mathcal{T} \cup \{T\}, \sigma \mapsto \sigma']}$	If an execution of a transaction T proceeds to completion without exceptions, then the transaction is added to the multiset of completed transactions.
Tx-Failure	$\frac{LookupStmt(M, pc) = \textbf{throw},\ f \leftarrow \langle \ell, Id, M, pc, \langle i, \bot\rangle\rangle,\ c = \langle f.\epsilon, \sigma\rangle}{c \rightsquigarrow c[f.\epsilon \mapsto \epsilon]}$	Since an exception of a callee frame does not propagate upwards to the caller, a transaction can fail only when the starting method frame throws an exception. In such an event, the list of completed transactions remains the same while the stack frame is made empty.
Add-block	$\frac{\langle\langle \mathcal{T}, \sigma\rangle, BC\rangle, \langle \epsilon, \sigma\rangle}{\langle\langle \mathcal{T}, \sigma\rangle, BC\rangle \rightarrow \langle\langle \epsilon, \sigma\rangle, BC.\mathcal{T}\rangle}$	The list of completed transactions is committed to the blockchain by this rule. We have explicitly ignored the broadcast and verification of proof-of-work as they have no affect on the global or blockchain state.

Table 1: Semantic rules for abstract assertion language. Note that M can be obtained by calling *LookupCode(Id, σ)* and the statement about to be executed can be obtained by the function *LookupStmt(M, pc)* where pc is pointing to the next statement to be executed.

realized by performing a taint-analysis to determine the set of locations where Sub and Obj are conjunctively used. This set is further refined by choosing only those control locations where the specified operation is invoked. The final locations are the ones where $Cond$ must be asserted based on $trig$.

C. Soundness

The proof for soundness of the translation from a Solidity contract to our abstract language with assertions (corresponding to policy predicates), and finally into LLVM bitcode entails the following steps. First, we discuss that translation of Solidity code into our abstract language does not affect semantic behavior. Second, we argue that a conservative placement of asserts does not affect the soundness of the approach. Third, we reduce the problem of policy confirmation to a state reachability problem. Fourth, we provide a definition of state reachability in the context of a Solidity program. Fifth, we demonstrate that by ensuring state reachability on an over-approximate version of the program, we do not miss on any program behaviors. Lastly, we argue that since our translation from this over-approximate Solidity program to LLVM bitcode is a faithful expression-to-expression translation, our overall soundness modulo the decision procedure is preserved.

(I) TRANSFORMATION FROM SOLIDITY TO ABSTRACT LANGUAGE. Since Solidity maps semantically to our abstract language (per § IV-A), this translation preserves the semantic behavior of the original program. While a formal argument about the semantic equivalence of Solidity and our abstract language is desirable, we omit it in the interest of space.

(II) EFFECT OF TAINT ANALYSIS ON SOUNDNESS. Note that taint-analysis, which is required to determine the locations at which to assert the predicates, is conservative. Thus, it may potentially insert asserts at multiple locations. While such extraneous asserts may introduce false positives, they do not affect false negatives. Hence, the approach is sound.

(III) SEMANTIC INTERPRETATION OF POLICY CONFIRMATION. Since the policies are restricted to quantifier-free FOL, policy confirmation can be reduced to a *state reachability* problem, i.e., does there exist a state reachable from the start state at which the policy does not hold? Formally, a policy ϕ holds on a program P when $N \models \phi$, where N is a formal representation of program P as a state-transitioning finite automaton. Note that `assert(exp)` in the abstract language has different semantics than the assert statements in high-level languages such as C (per Table 1).

(IV) ASSERTION SAFETY IN A PROGRAM IMPLIES POLICY CONFIRMATION. Consider a program $\hat{P}$ (corresponding to a Solidity program P) in our abstract language but without any `asserts` or `havocs`. Let $B_{\hat{P}}$ be the set of behaviors of $\hat{P}$ described as $B_{\hat{P}} = \{s \mid \forall\ s_0 \in I, s \in Reach(s, s_0)\}$ where I is the set of initial states and the relation $Reach(s,\ s')$ is true iff $s \rightarrow^* s'$. Consider the translation $\hat{P} \rightarrow \hat{P}'$, where $\hat{P}'$ has `asserts` inserted according to the rules governed by the compilation of policies into `asserts` as described above.

Lemma 1: Assertion safety in $\hat{P}' \Leftrightarrow$ Assertion safety in $\hat{P}$.

Proof: The semantic rule for the `assert` statement indicates no change in the data state of the program except the change

of the program counter. `asserts` are relevant only as tags for the underlying verifier to generate verification conditions. Thus, it follows that $B_{\hat{P}} = B_{\hat{P}'}$, which implies that assertion safety in $\hat{P}'$ is equivalent to assertion safety in $\hat{P}$. ■

(V) SOUNDNESS VIA OVER-APPROXIMATION. Consider now a translation of program $\hat{P}'$ to $\hat{P}''$, where definition of global variables is replaced by `havoc` statements.

Lemma 2: Assertion safety in $\hat{P}'' \Rightarrow$ Assertion safety in $\hat{P}'$.

Proof: It is clear that any `havoc(x)` statement in the program expands the domain of legitimate values that the variable x can take to the type-defined domain of that variable. Thus, $B_{\hat{P}'} \subseteq B_{\hat{P}''}$. This relation implies our lemma statement. ■

Theorem 1: Assertion safety of $P'' \Rightarrow$ Assertion safety of P.

Proof: The proof follows from Lemma 1 and 2. ■

(VI) SOUNDNESS OF METHODOLOGY. Let P''' be a faithful translation of the over-approximate program P'' into LLVM bitcode. Table 2 lists the details of our semantically equivalent expression-to-expression translation strategy. In other words, $\hat{P}'' \cong \hat{P}'''$. Like prior art [54], a formal proof for soundness of the translation strategy is outside the scope of this work.

Thus, the overall soundness of our methodology logically follows from Theorem 1 and $\hat{P}'' \cong \hat{P}'''$, and the established soundness of ZEUS' underlying decision procedure [61].

D. Symbolic Model Checking via CHCs

ZEUS uses prior art [61] to emit verification conditions as CHCs for the translated program $\hat{P}'''$. The strength of the CHC representation enables it to interface with a variety of SMT-based solvers and off-the-shelf model checkers [56], [63].

E. End-to-end example

Fig. 12 presents an end-to-end example complete with all program transformations. The Solidity snippet sends `msg.value` to the address `msg.sender` and updates its local balance by subtracting the same from `bal[msg.sender]`. The example policy checks that the `send` invocation must satisfy the condition that the balance for the user must be greater than the value to be sent. ZEUS extracts the predicate from the policy condition and places it as an assert in the Solidity code. Subsequently, it transforms it into the abstract language. Lastly, ZEUS translates this program into LLVM bitcode.

Solidity code

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

Policy

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

Abstract Program

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

LLVM bitcode

```
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":                                             ; preds = %"64"
  call void @__VERIFIER_error()
  br label %"75"

"75":                                             ; preds = %"74", %"64"
  %__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
}
```

Fig. 12: An end-to-end example.

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

Fig. 13: Fairness policy for example in Fig. 10.

V. IMPLEMENTATION

We implement a prototype of ZEUS as described in § IV. We implement the policy builder and the Solidity to LLVM bitcode translator in C++ using the AST of the smart contract derived from the Solidity compiler `solc` [41]. The policy builder and the translator (including the LLVM passes for bug detection) required around 500 and 3000 LOC respectively.

For ease of implementation, we leverage Seahorn [61] as our symbolic model checking backend for verification of policies. Instead of building the verifier from scratch, we determined that Seahorn provides us with an off-the-shelf implementation of generating verification conditions using CHCs over LLVM bitcode. Furthermore, use of existing tools that have been tested for bugs and fine tuned for performance, both of which are critical for verifiers, helps reduce ZEUS's false alarms and improve verification times. However, as will be shown later in § VI-D, ZEUS is not tied to Seahorn; it can be used with any other verifier that operates upon LLVM bitcode, such as SMACK [72] or DIVINE [52].

A. Policy Builder

ZEUS extracts the identifier information (for subjects and objects) from the corresponding AST node in the `solc` parser to build the policy. The operation is extracted from the function call node in the AST, while the predicates are extracted from the conditionals in the node representing binary operations.

Algorithm 1 briefly lists the steps to build the policy specification. Specifically, ZEUS runs a taint analysis pass over the contract code with sources ($\mathbb{S}$) as contract- and runtime-defined global variables in Solidity [48]. The sinks ($\mathbb{F}$) are invocations to external APIs calls, such as `send` or publicly invocable functions. ZEUS also captures control flow conditionals, or path predicates, for all flows originating at

```
POLICY_BUILDER(ℂ)
Input: ℂ: Smart contract source code.
Output: ℙ: Policy specification
𝕊𝕊 := TAINT_ANALYSIS(ℂ); sub := Get_Subjects_From_User(𝕊𝕊)
α := 𝕊𝕊[sub]; obj := Get_Objects_From_User(α)
β := α[obj]; ops := Get_Operations_From_User(β, obj)
γ := β[ops]; con := Get_Predicates_From_User(γ, ops)
trigger := Get_Trigger_From_User(); result := Get_Result_From_User()
ℙ := Create_Policy(sub, obj, ops, con, trigger, result)
return ℙ

TAINT_ANALYSIS(ℂ)
Input: ℂ: Smart contract source code.
Output: Ψ: Set of state space tuples.
Initialize: 𝕊: {Global variables}, 𝔽: {Publicly visible functions}, Ψ := {}
foreach ((ρ ∈ 𝕊) ∧ (ϕ ∈ 𝔽)) do
    ℂ:= {}; ℙ:= {};
    τ := GET_ALL_TAINTED_STATEMENTS(ρ, ϕ)
    foreach (pc ∈ τ) do
        obj := Get_Objects(pc)
        ℂ := ℂ ∪ obj
        if (Is_Conditional(pc)) then p := Get_Predicate(pc); ℙ := ℙ ∪ p ;
    end
    Ψ := Ψ ∪ (ρ, ℂ, ϕ, ℙ)
end
return Ψ
```

Algorithm 1: Steps to build the policy specification.

the sources and terminating at the sinks. The output of the taint analysis pass is a set of tuples consisting of the source, the objects, the sink and its corresponding path predicates. ZEUS then lists the set of all available taint sources, i.e., the globals and the environment variables, from which a user selects the subject to be tracked. It then filters the results from the taint pass that reduces the search space to tuples containing at least one of the subjects selected by the user. ZEUS then prompts the user to select the object(s), following which it further prunes the tuple list. It then displays the list of potential invocations that involve at least one or more of the subjects or objects. Upon further selection, ZEUS lists the available predicates encountered along the source to the sink. The user can compose these predicates (or specify his own) using boolean operators to form the condition in the policy, and indicate whether they are checked as a pre- or post-condition. Finally, the user indicates in the result tag whether the specification determines a violation or accepted behavior.

Fig. 13 lists the fairness criteria for the example shown in Fig. 10. Correctness polices use a similar template with the operation specifying the bug class to be detected.

B. Solidity to LLVM Bitcode Translator

ZEUS takes in a smart contract and passes it through the translator to generate its LLVM bitcode along with the debug symbol information. Subsequently, for ease of implementation, it reads the policy specification and rewrites the bitcode (instead of the Solidity source code) to inject assert conditions for the predicates as per the trigger attribute in the specification. Most Solidity statements and expressions have the same semantics as their C/C++ counterparts. We use the rich LLVM APIs to generate semantically correct bitcode while traversing the AST during code compilation (per Table 2). We handle expression translation using the standard LLVM APIs.

ENSURING SOUNDNESS. In Solidity, execution of a public function constitutes a transaction, which can be reordered at the miner. To be sound, ZEUS must correctly reason about all possible execution orders and control paths in the program.

(**1**) **Execution order**: Since there can be arbitrarily many parallel invocations of a contract's public functions with no global invocation order enforced, modeling these infinitely many execution orders is not possible. We observe that for six of the seven classes (except transaction-order dependence), bug detection is intra-transaction. In other words, the verifier needs to reason about bugs within one call chain. The ordering of transactions does not impact bug detection within a transaction. Detection of transaction-order dependence involves detecting writes and subsequent reads to sensitive global variables across a pair of transactions. Reasoning about pairs of functions suffices, because a minimal instance of transaction-order dependence must manifest across at least two function invocations. Ordering of remaining functions is immaterial for detecting the bug. Thus, for a contract with n publicly available functions, ZEUS must reason about $O(n^2)$ possible orderings. ZEUS generates a set of `main` functions, which act as the entry point for verification. Specifically, a `main` function is a harness that `havoc`s all the global state before invoking a publicly defined function. For transaction-order dependence, ZEUS `havoc`s the global state and invokes pair-wise permutations of all public functions from within a `main` function. Further, ZEUS `havoc`s the entire global state upon invocation of any member of the `call` family.

(**2**) **Path traversal**: In Solidity, global variables hold state across executions. Modeling and reasoning all such states in one static analysis execution is impractical. Thus, ZEUS abstracts the values of all globals in the contract, including the block and transaction state variables, to the entire data domain corresponding to the data type. For example, a global of type `uint256` is modeled as having an integer domain, and can take values anywhere between 0 and `INT_MAX`. For any concretely defined starting values, ZEUS automatically `havoc`s them to explore the entire data domain. Keeping the initial value constant does not reason about all possible executions since the value may be incremented in subsequent contract executions and may lead to potential exploitability in the future. Thus, a single static execution suffices to analyze all possible control paths.

MODELING SOLIDITY SYNTAX. ZEUS supports complex Solidity syntax, including inheritance, external functions, tuple expressions, modifiers, operations over nested `struct` definitions, iteration over maps and arrays, and memory allocation/de-allocation. We discuss a few of them below:

(**1**) **Inheritance**: Solidity allows multiple inheritance amongst the contracts. ZEUS follows the same logic used by Solidity to implement inheritance, i.e., the base contract on the extreme right in the declaration overrides all functions previously declared by other base contracts. Specifically, ZEUS generates the LLVM bitcode per contract, and then follows the said order to patch the overriding functions visible in the derived class.
(**2**) **External functions**: Solidity allows one contract to call into another contract. Furthermore, Solidity mandates that all external functions only take in primitives as input and the returns are also of primitive types [36]. While ZEUS cannot resolve such functions at compile time, it over approximates their behavior for soundness and assumes that these external functions return a non-deterministic value. This non-determinism soundly models the execution semantics of these external functions, i.e., the return can take any value.
(**3**) **Arrays**: ZEUS does not implement dynamic arrays in

AST Node	Abstract	LLVM API	Comments
`ContractDefinition`	contract@Id{...}	`Module`	Creates a new module, sets the data layout, generates the definition of global variables and functions, and writes a main function which serves as the driver.
`EventDefinition`	function@Id(l:T){S}	`FunctionType`, `Function`	Creates a new function with the return type void, the arguments type as specified in the event, and inserts it in the given module.
`FunctionDefinition`	function@Id(l:T){S}	`FunctionType`, `Function`	Creates a new function with the given return type, arguments and the body.
`Block`	{S}	`BasicBlock`	Creates a BasicBlock inside the LLVM IR, sets the insertion point to this block, and iterates over the statements to generate the IR for each one.
`VariableDeclarationStatement`	(l:T)*	`CreateStore`, `CreateExtOrTrunc`	Iterates over all the variables declared in this statement, allocates the variable using the VariableDeclaration node, and stores the initial value (or a default value) to the allocated space. (after sign / zero extension / truncation if needed).
`VariableDeclaration`	(l:T)	`GlobalVariable`, `CreateAlloca`	For a global variable, uses the GlobalVariable API to define a global variable, otherwise, allocates space using the alloca instruction.
`Literal`	ℓ	`ConstantInt`	Allocates a constant value for the various types of integers of varying widths.
`Return`	`return` e	`ReturnInst`, `CreateExtOrTrunc`, `CreateGEP`	Uses other AST nodes to generate the expression to return, dereferences it, extends / truncates the value, and returns the value using the return instruction.
`Assignment`	l := e	`CreateExtractValue`, `CreateExtOrTrunc`, `CreateLoad`, `CreateStore`, `CreateBinOp`	Generates the right hand side of the expression. A tuple is unpacked. For compound assignments, the corresponding binop instruction is created, and the result is sign/zeroextended/truncated and stored in the left hand side.
`ExpressionStatement`	e		Calls the ASTNode for the Expression to generate the LLVM IR.
`Identifier`	Id	`ValueSymbolTable`, `GlobalVariable`, `getFunction`	Checks for the identifier inside the local variables, global variables or the functions written in the contract, and returns the appropriate LLVM object.
`IfStatement`	`if` e `then` S `else` S	`BasicBlock`, `CreateBr`, `CreateCondBr`	Generates the condition variable inside the current block, creates a conditional branch, and branches to either the true or the false branch. In the absence of the false branch, uses the branch instruction to fall through.
`FunctionCall`	`goto` or `post`	`CreateExtOrTrunc`, `CreateCall`, `Function`	Generates the arguments for the function call, fixes their type according to the solidity semantics, and creates a call to the required function.
`WhileStatement / ForStatement`	`if` e `then goto` l `else` S	`BasicBlock`, `CreateCondBr`	Generates the conditional variable, the body of the loop, and a branch instruction.
`StructDefintion`	T	`StructType`	Generates a structure of the same type as in the solidity contract.
`Throw`	`throw`	`Function`, `CreateCall`	Calls the system's exit function.
`Break / Continue`	`if` e `then goto` l	`CreateBr`	Keeps a stack of the break/continue tags and branches to the appropriate label.

Table 2: Expression to expression translation from Solidity code to LLVM bitcode.

LLVM bitcode but uses a static array with large length. This minor tweak preserves the semantic meaning of the contracts and makes them amenable for verification. All strings and byte accesses are also modeled as arrays with integer domain.

(**4**) **Rational Ether**: Solidity does not implement floating point arithmetic. It instead uses rational numbers to implement fractional payouts. ZEUS converts such rational payments into lower monetary units to allow integer arithmetic in LLVM bitcode. For example, ZEUS converts 1/4 ether to 250 finney.

HANDLING LLVM OPTIMIZATIONS. LLVM's optimizer can run aggressive passes eliminating any non-side affecting variables and function calls. However, this can adversely impact the verification result. For example, if return values from invocations such as `send` are not used, both the `send` call and the return value are optimized out. This optimization causes problems in detecting scenarios described earlier in § III-A and § III-C. Further, the verifiers may invoke their own optimization passes that may mess with the LLVM bitcode translation from Solidity code. In an effort to remain faithful to the semantics envisioned by the contract writer, ZEUS creates a global variable for each external function return value and enforces no optimization on all functions.

LIMITATIONS. Our prototype of ZEUS has a few limitations across policy specification, translation and verification.

(**1**) Fairness properties involving mathematical formulae are harder to check. For example, 25% can be represented in several different forms. ZEUS depends on the user to accurately define policies that involve such mathematical representations.

(**2**) ZEUS is faithful to most Solidity syntax. However, constructs like `throw` and `selfdestruct` which have no exact LLVM bitcode transformation are modeled as a program exit. Further, runtime EVM parameters such as gas consumption cannot be precisely computed at the source level. Thus, ZEUS is overly conservative in its runtime behavior modeling and does not explicitly account for these parameters.

(**3**) ZEUS does not support virtual functions in contract hierarchy, i.e., use of `super` keyword, which resolves the function call at runtime, dependent on the final inheritance graph. We manually analyzed the 23 such contracts in our dataset and resolved the `super` linkages.

(**4**) Solidity's `assembly` block allows use of EVM bytecode alongside regular Solidity statements. Even though real-world contracts rarely use `assembly` (only 45 out of 22,493 contracts in our data set use it), ZEUS is conservative and does not analyze contracts with an `assembly` block.

(**5**) ZEUS supports verification of safety properties, i.e., state reachability expressible via quantifier-free logic with integer linear arithmetic. Verification of liveness (i.e., something good must eventually happen) requires support for linear temporal logic, and is currently not supported by ZEUS. Extending ZEUS to support other kinds of properties such as trace- or hyper-properties does not require changes to the core design and we leave it for future work.

```
(1)   mapping(address => uint) private userBalances;
(2)   function withdrawBalance'() {
(3)     uint amountToWithdraw = userBalances[msg.sender];
(4)     if (amountToWithdraw > 0) {
(5)       assert(false);
(6)       msg.sender.call(userBalances[msg.sender]);)
(7)       userBalances[msg.sender] = 0;
(8)     }
(9)   }
(10)  function withdrawBalance() {
(11)    uint amountToWithdraw = userBalances[msg.sender];
(12)    if (amountToWithdraw > 0) {
(13)      withdrawBalance'();
(14)      msg.sender.call(userBalances[msg.sender]);
(15)      userBalances[msg.sender] = 0;
(16)    }
(17)  }
```

Fig. 14: Same-function reentrancy detection for example in Fig. 2.

C. Handling Correctness Bugs

ZEUS provides verification for the correctness issues described in § III-A and § III-C. We discuss the verification logic for them as implemented in several LLVM passes below.

(**1**) **Reentrancy**: Reentrancy in Solidity can happen via the `call` method. `send` only invokes the default function with limited gas for logging purposes. ZEUS handles same-function reentrancy by first cloning the function under consideration, and inserting a call to the clone before the invocation to `call`. Fig. 14 shows the patched function for the example in Fig. 2. Note that ZEUS ensures that the patch is done within the same basic block so as to ensure that if the cloned function is called, then the invocation to `call` is also made. Further, we also assert false before the `call` code. If the verifier finds a path leading to this assert, it indicates a bug.
Cross-function reentrancy can be handled similarly by patching different functions. However, it is not scalable due to state space explosion even with small number of functions. We leave efficient detection of cross-function reentrancy for the future.
(**2**) **Unchecked `send`**: Detection of unchecked `send` bug requires identifying any subsequent accesses to global state variables in case of a failed `send` call. ZEUS initializes a global variable `checkSend` to `true` and takes its conjunction with return value from every `send` operation. For every subsequent write of a global variable, an LLVM pass automatically inserts an `assert(checkSend)` in the bitcode.
(**3**) **Failed `send`**: Recall that the aim here is to prevent reverting the transaction by not invoking `throw` on a failed `send` call. The detection of this bug leverages the same check as the unchecked `send` scenario, but places the assertion ahead of the `throw`. While the `throw` encountered may be due to some other condition in the code, the counterexample indicates a possibility of reverting the transaction due to control flow reaching a `throw` on a failed `send`, not necessarily the immediate `throw` associated with the `send`.
(**4**) **Integer overflow/underflow**: An LLVM pass implements the overflow/underflow detection checks for all arithmetic operations [20], consistent with Solidity's semantics, i.e., an overflow/underflow for integers causes a wrap around.
(**5**) **Block/ Transaction state dependence**: Bug detection for these classes requires context sensitive information. For example, block state dependence requires determining if block variables, such as `timestamp`, flow into `send` or `call`. We implement our own taint analysis pass over LLVM bitcode and use symbolic model checking to eliminate infeasible paths.

Category	#Contracts	#LOC (K)		#Send/Call	#Ext. Calls
		Source	LLVM		
DAO	140	2.8	24.3	252	350
Game	244	23.3	609.2	851	16
Token	290	25.2	385.9	311	271
Wallet	72	10.8	105.9	186	6
Misc.	778	47.6	924.3	1102	498
Total	1524	109.7	2049.6	2702	1141

Table 3: Characterization of the dataset.

Note that Solidity's lack of pointer arithmetic, unlike C/C++, eases our taint tracking implementation.
(**6**) **Transaction order dependence**: We detect transaction order dependence by determining potential read-write hazards for global variables that can influence Ether flows. Specifically, we taint all global variables that are written to and then determine if this taint flows into a `send` or `call`.

VI. EVALUATION

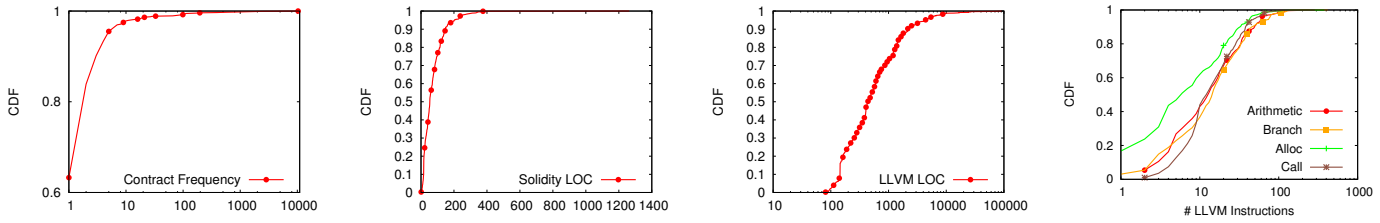
EXPERIMENTAL SETUP. All experiments were performed atop an IBM System x3850 X5 machine having 4 Intel Xeon E7-4860 CPUs at 2.27 GHz with 10 cores each and 2 threads/core, and 512 GB of RAM, running 64-bit Ubuntu v14.04. We built our Solidity to LLVM bitcode translator over `solc` [41], which is compatible with LLVM 3.6. We used a stable build of Seahorn [37] (snapshot of March 31^{st}) as our verifier and set a timeout threshold of 1 minute. For comparisons with Oyente, we use their snapshot as available on April 15^{th}, and keep a timeout duration of 30 minutes [68].

A. Data Set

We periodically scraped Etherscan [18], [19], Etherchain [12] and EtherCamp [11] explorers over a period of three months and obtained source code for 22,493 contracts at distinct addresses. We discounted 45 contracts that had an `assembly` block in their source, and obtained 1524 unique contracts (as per their `sha256` checksum). In the remainder of the section, we present results only for these unique contracts, unless specified otherwise.

We analyzed all unique contracts and classified them under five categories. "DAO" enlists all contracts that involve DAO-style investment. Contracts involving games and decentralized gambling, are clubbed under "Game". "Token" contracts implement the standard tokens for designing financial instruments. "Wallet" contracts implement a user wallet. All other contracts are listed under "Misc". Table 3 summarizes their characteristics. We note that contracts in the "DAO" and "Token" categories leverage a lot of external functionality, unlike "Game" and "Wallet" that appear to be self-contained. "Game" contracts involve significantly more number of `send`/`call` invocations than any other category, indicating a lot of Ether transfer between participants. "Misc." contracts account for half of the unique contracts, indicating the diverse nature of contracts available on Ethereum. Lastly, the generated LLVM bitcode is an order of magnitude more than the source LOC, since it is unoptimized.

Fig. 15a shows the frequency of duplicates across our data set. We observe that less than 5% contracts have more than 10 duplicates. For example, one wallet contract [49] in our data set was duplicated 10.3K times. Further, the frequency of



(a) Frequency of duplicate contracts. **(b)** Source LOC across contracts. **(c)** LLVM LOC across contracts. **(d)** Frequently used LLVM operations.

Fig. 15: Data set characterization.

duplicate token contracts at different addresses was also high. In other words, contracts providing useful functionality are more likely to be duplicated. Fig. 15b plots the source LOC in Solidity for the unique contracts. The total source LOC was over 111K, while mean and median were 74 and 54 LOC respectively. The largest contract we analyzed had 1405 LOC. However, over 90% of the contracts have 200 LOC or less.

ZEUS generates more than 1K LOC of LLVM bitcode for around 30% of the contracts, with maximum being 91,338. (per Fig. 15c) The mean and median bitcode per contract were 1354 and 439 respectively. In all, ZEUS verified over 2M lines of bitcode. Note that our LOC measurements did not include any blank lines or comments. Lastly, we plot the frequently occurring operations in the bitcode. Fig. 15d presents the results for top four classes of operations, besides memory `load`, `store` and `GetElementPtr`. We observe that "arithmetic" operations are as frequent as "comparison", "call" and "alloc" operations.

B. Results with Solidity-based Smart Contracts

CORRECTNESS AND MINER'S INFLUENCE. We evaluate ZEUS for all 1524 unique contracts for issues described in § III-A and § III-C, and compare with Oyente for the common bug classes, i.e., reentrancy, unchecked `send`, block-state dependence and transaction-order dependence. We note output for each scenario as either "Safe" or "Unsafe" (i.e., there is a potential issue). In case the verifier throws an error or quits before the timeout, we categorize the contract as "No Result". All other cases are categorized under "Timeout".

Note that Solidity contracts are small and have few local/global variables, which makes it tractable to capture/track dependencies. Further, since no tools exist to ascertain the ground truth, we manually validated each result to determine the set of false positives and negatives. A false positive occurs when ZEUS returns "Unsafe" but the contract is actually "Safe", while a false negative occurs when ZEUS returns "Safe" when the contract is actually "Unsafe". The false alarm rate is the ratio of false positives over the total results returned, i.e., both "Safe" and "Unsafe". Table 4 lists the results for both ZEUS and Oyente. The entire set of results are also available at https://goo.gl/kFNHy3.

(**1**) **Reentrancy:** ZEUS detects 54 contracts as vulnerable to reentrancy. More importantly, it gives 0 false positives and negatives, primarily due to its callee function patching mechanism (per § V-C). In contrast, Oyente reports 265 contracts to be susceptible to reentrancy. Since Oyente does not distinguish between `send` and `call` functions at the bytecode level, it also considers reentrancy on `send`, which cannot occur (recall § III-A). Hence, it reports a large number of unsafe contracts, with a high false alarm rate of over 31%.

(**2**) **Unchecked `send`:** ZEUS reports 324 contracts affected by the unchecked `send` bug, with 3 false positives and 0 false negatives. We analyzed the offending contracts and observed that ZEUS's over-approximation in `havoc`ing all globals to traverse control flow not intended by the contract developer and detect bugs along those paths. In contrast, Oyente marked 112 contracts as unsafe with no result in 203 contracts. Further, it reports a high false alarm rate of around 7.5%.

(**3**) **Failed `send`:** ZEUS detects 447 contracts vulnerable to the failed `send` bug with 0 false positives and negatives.

(**4**) **Integer overflow/underflow:** Smart contracts involve a lot of arithmetic operations (per Fig. 15d), and contract writers typically do not check for overflow/underflow conditions. This is corroborated by the fact that 1095 of the 1524 contracts (or around 72%) are vulnerable to this bug.
However, ZEUS also reports 40 false positives (i.e., a false alarm rate of 2.7%). We manually analyzed all false positives and observed that they stem due to ZEUS `havoc`ing all globals. For example, in one contract the percentage payout was declared a global with a fixed value. However, ZEUS initializes it to the entire data domain, following which an operation using the payout variable causes the operation to overflow. A better program analysis (or abstract refinement) of smart contracts can help weed out such cases, where it is not required to assign the entire data domain to a global variable.

(**5**) **Transaction state dependence:** ZEUS found 1513 contracts out of 1524 to be safe, while only 8 contracts were deemed unsafe, with 0 false positives and negatives. ZEUS was successfully able to detect the vulnerability as described in Fig. 6 in the contract `LittleEthereumdoubler` [17].

(**6**) **Block state dependence:** ZEUS found 250 contracts to be vulnerable, with 0 false positives/negatives. In contrast, Oyente marked 15 contracts as unsafe, and failed to provide results for 711 contracts (either due to timeout or no result). ZEUS is conservative and considers all `block` parameters can be modified, while Oyente considers only `timestamp`.
We observed that solWallet [49], which has over 1.4 million Ether in balance, is also susceptible to the block state dependence bug. It uses `now` (an alias for `block.timestamp`) that can easily be tampered with by the miner [14]. For example, a miner can use a value of `now`, which can lie anywhere between the current timestamp and 900 seconds in the future, and allow monetary transactions (close to the end of the day) even when its daily limit has been exhausted.

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

Table 4: ZEUS's evaluation and comparison with Oyente [68].

(**7**) **Transaction order dependence:** ZEUS reported 607 contracts (or 39.8%) as unsafe, with 16 false positives, and 0 false negatives. The false positives stem from `havoc`ing the globals leading to traversal of paths not intended by the developer. In contrast, Oyente reported 129 contracts as unsafe along with no decision for 707 contracts. It reported a false alarm rate of 14.2%; an order of magnitude more than ZEUS.

SUMMARY. The above results enable four key observations:

- 21,281 out of 22,493 contracts (or 94.6%) containing more than $0.5 billion worth of Ether are vulnerable to one or more bugs. Across the unique contracts, 1194 out of 1524 contracts were found to be vulnerable to one or more bugs.
- ZEUS's use of abstract interpretation along with symbolic model checking for verification makes it sound. We observed zero false negatives for all the seven bug classes. The low false alarm rate can be further mitigated by improved program analysis to weed out scenarios not intended by the developer.
- Use of CHCs enable quick verification, with only 44 out of 1524 contracts (or 2.89%) timing out in at least one bug class. ZEUS's timeout threshold is fairly low at just 1 minute.
- Oyente is neither sound nor complete, and reports a high false alarm rate for three of its four bug classes. Further, it times out or gives no result for 711 contracts (or 46.7%) in our dataset. These numbers are consistent with their published dataset, where almost half the tests gave no result [33].

DISCUSSION. ZEUS determines a contract as either safe or unsafe, i.e., if a contract is vulnerable in principle or not. An unsafe result does not guarantee a trivial exploit. For example, several contracts are vulnerable to integer overflow because they do not check for the bounds. Thus, ZEUS marks them as unsafe. Even though possible in principle, these contracts may not be susceptible to an immediate exploit, say when the payouts use `uint256` for calculation. Similarly, contracts using `timestamp` for control flow, may not be affected immediately, but a path may exist with a certain value of the `timestamp` that affects the Ether flow in a `send` invocation.

FAIRNESS. We select representative contracts from the four classes (per § VI-A) and apply contract specific properties along with a common fairness policy across all contracts.

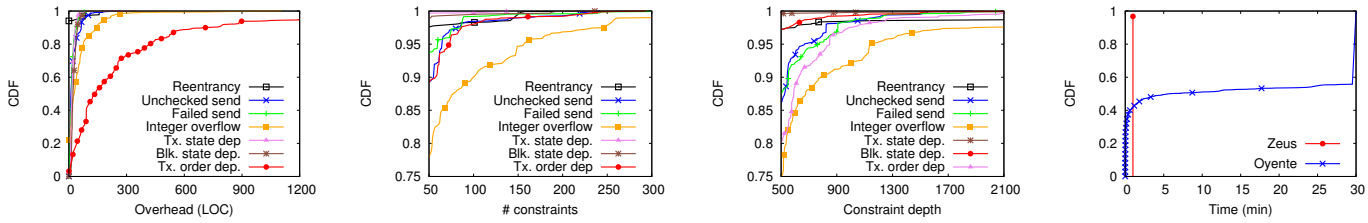
(**1**) **DAO:** CrowdFundDao [9] implements a DAO scheme allowing investors to choose when to pay and withdraw their funds. We implemented two policies: (a) blacklisted developers cannot participate in the scheme, and (b) the investment must be more than a threshold limit. ZEUS determined that none of these checks were encoded in the contract.
(**2**) **Game:** DiceRoll [10] is a dice game where players join a game by placing a bet. We implemented a policy that the number of dice rolls for a player must be limited. We observed that the game did not cap the number of dice rolls per user.
(**3**) **Token:** Tokens, such as StandardToken [42], are used to implement financial instruments. While most of them consider integer underflow possibilities on the sender side, it is important for them to consider whether token transfer could result in overflows on the receiver. We implemented this policy on several contracts and observed that some of them, such as Campaign [5], do not consider overflows at the recipient.
(**4**) **Wallet:** Wallet [49] implements several functionalities for users, including a daily withdrawal limit. We check for two policies: (a) a user cannot send money to themselves, and (b) there is a limit on the amount being transferred per transaction. We observed that both these policies report a violation.
(**5**) **Common policy:** Solidity provides `selfdestruct` to kill a contract and return its balance to an address. We check if the construct is invoked only by the owner. We observed that 284 out of 1524 contracts had this construct, with around 5.6% reporting a violation of the policy with no false alarms.

C. Performance

(**I**) **INSTRUCTION OVERHEAD.** ZEUS's Solidity to LLVM bitcode translator introduces checks for several bugs described in § III-A and § III-C. Fig. 16a plots a CDF of this instruction overhead due to additional LLVM bitcode LOC introduced per contract. We observe that ZEUS introduces less than 50 LOC for 97% of contracts across five of the seven bug classes. For integer overflow/underflow, ZEUS's checks account for less than 200 LOC in 95% of the contracts. However, detection of transaction order dependence incurs maximum overhead, with 20% contracts requiring over 500 LOC for the required checks.

(**II**) **ANALYSIS COMPLEXITY.** We measure the verification complexity by determining the number of rules generated and their depth per contract across the seven bug classes. Fig. 16b and Fig. 16c plot the results. We observe that 75% of contracts across all categories generate less than 50 rules with depth of around 700. Overall, integer overflow generates maximum constraint rules and depth across all categories, with a maximum of 1035 rules and depth of 277,345. This behavior is consistent with our observation that contracts have significant amount of arithmetic operations (recall Fig. 15d).

(**III**) **ANALYSIS TIME.** We determine ZEUS's verification time for each unique contract in our data set and compare against Oyente. Fig. 16d plots the CDF of the results. We observe that ZEUS takes a minute or less for verifying 97% contracts (as indicated by the vertical line). Only 44 contracts out of 1524 timeout or give no result for one or more bug classes. In contrast, Oyente returns results for only 40% contracts within one minute. Furthermore, it provides no result or timeouts (even after 30 minutes) for around 43% contracts.



(a) LLVM bitcode instruction overhead. **(b)** Verification rules generated. **(c)** Verification constraints generated. **(d)** Verification time in minutes.

Fig. 16: ZEUS's performance.

D. Case Studies

We now demonstrate ZEUS's versatility with other blockchain platforms and verification engines. First, we port a popular Solidity contract to Fabric [24]. Second, we use SMACK [72] to verify fairness policies for the said contract. We describe our experiences below.

Simple Dice [39] is a popular multi-player gambling contract, where players put in a minimum deposit (along with a fee) to play. The players have a 25% chance of winning the entire balance. Also, every 40*th* player wins the jackpot, which is considerably more than the balance. The fee and deposit rate can only be changed by the owner, and is publicly visible.

To validate the contract, we check the following five policies: (a) a minimum deposit is required to play the game, (b) every 40*th* player does win the jackpot, (c) only the owner can change the deposit and fee rate, (d) the owner cannot participate in the game, and (e) every player must have an equal chance of winning the jackpot, i.e., a player must not have multiple entries when playing for the jackpot.

(I) FABRIC. Smart contracts in Fabric can be written in high-level languages, such as GO and JAVA. Solidity's global variables that persist state across transactions are mapped on to the blockchain. In contrast, high-level languages do not have this support; the globals share state across functions calls. Fabric gets around this problem by defining a shim layer for each high-level language that exports APIs to allow smart contracts to explicitly manage state atop the blockchain.

We ported the Simple Dice contract to GO and linked it against Fabric's mock-stub [25]. However, we noticed that the mock-stub takes strings as input and converts them to the required data types using standard GO libraries. Since Seahorn does not have support for strings, we fixed the mock-stub to take integers as input. While policy specification is automated for Solidity, we manually edited the GO code and placed the correct assertions along the required program paths, corresponding to the above mentioned policies. We then leveraged `llgo` [27] to generate LLVM bitcode for Simple Dice and verified the policies with Seahorn.

(II) SMACK. To leverage SMACK as the verifier of choice, we had to make three key modifications. First, since SMACK supports integer operations up to 64 bytes [30], we had to port our Solidity to LLVM bitcode translator to work with 64 bytes. Note that Solidity supports integers of length 256 bytes. Second, SMACK, unlike Seahorn, requires developers to use different APIs to request for the entire integer domain [40]. For example, one has to use `__VERIFIER_nondet_ushort` for modeling integer domains for `unsigned short`, and `__VERIFIER_nondet_uint` for `unsigned int`. Third, the APIs for invocation to the verifier are different between SMACK and Seahorn. Overall, we required around 50 lines of modifications to ZEUS to make it compatible with SMACK. With our SMACK-compatible ZEUS, we verified the five fairness policies for Simple Dice as described earlier.

Our experiences with both Fabric and SMACK suggest that it is easy to extend ZEUS to other blockchain platforms and verifiers with only minor changes.

VII. RELATED WORK

SMART CONTRACT BUG DETECTION. We now compare and contrast ZEUS with related work in the area of smart contract bug detection, apart from Oyente [68], Bhargavan *et al.* [54] and Why3 [22], [50], which we have discussed earlier. Delmolino *et al.* [58] document several classes of mistakes when developing contracts, suggest ways to mitigate these errors, and advocate best practices for programming smart contracts. In contrast, ZEUS presents a formal verification framework for smart contracts that enables users to build and verify correctness and fairness policies over them.

POLICY SPECIFICATION. Naccio [60], PoET/Pslang [59] and Polymer [53] enable policy specification for security properties. Like prior work, ZEUS ensures that the verification policy is defined separately from the main application. This separation of logic makes it easier to understand, verify, and modify the security policy. XACML [51] defines a declarative fine-grained, attribute-based access control policy language that inspires ZEUS's syntax for policy declaration.

VIII. CONCLUSION

We present the design and implementation of ZEUS—a framework for analyzing safety properties of smart contracts. ZEUS leverages abstract interpretation and symbolic model checking, along with the power of CHCs to quickly ascertain the verification conditions. We build the first Solidity to LLVM bitcode translator to automatically insert verification conditions given a policy specification. Our evaluation with over 22.4K Solidity smart contracts indicates that about 94.6% of them (with a net worth of more than $0.5 billion) are vulnerable. ZEUS is sound (with zero false negatives) and significantly outperforms Oyente for contracts in our data set, with low false positive rate and an order of magnitude improvement in time for verification.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments. We are also grateful to Prasad Naldurg for his feedback on an earlier draft of the paper.

REFERENCES

[1] "Analysis of the DAO exploit," http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[2] "Auction house," https://git.io/vFAl6.

[3] "Blockchain investment in 2016," https://www.cryptocoinsnews.com/pwc-expert-1-4-billion-invested-blockchain-2016/.

[4] "Buggy contract," https://live.ether.camp/account/dfa42284475636ecc1e04f519b075ec7f1e04f48/contract.

[5] "CampaignToken," https://etherscan.io/address/0xa0388ffb2a3c198dee723135e0caa423840b375a.

[6] "Computation in Fallback Function," https://ethereum.stackexchange.com/questions/5992/how-much-computation-can-be-done-in-a-fallback-function.

[7] "Contest contract," https://etherscan.io/address/0x98086130278fe48f2f0330e330df6ed6c91ce4f7#code.

[8] "Cross-function Race Condition," https://git.io/vFA4y.

[9] "CrowdFundDAO," https://live.ether.camp/account/9b37508b5f859682382d8cb6467a5c7fc5d02e9c/contract.

[10] "DiceRoll," https://ropsten.etherscan.io/address/0xb95bbe8ee98a21b5ef7778ec1bb5910ea843f8f7#code.

[11] "EtherCamp," https://live.ether.camp/.

[12] "Etherchain," https://www.etherchain.org/contracts.

[13] "Ethereum," https://www.ethereum.org/.

[14] "Ethereum Block Protocol," https://git.io/vFA8I.

[15] "Ethereum Blog," https://blog.ethereum.org/2016/06/10/smart-contract-security/.

[16] "Ethereum Contract Security," https://git.io/vFA8a.

[17] "Ethereum Doubler," https://etherscan.io/address/0x83651a62b632c261442f396ad7202fe2a4995e3a#code.

[18] "Etherscan," https://etherscan.io/accounts/c.

[19] "Etherscan - Ropsten," https://ropsten.etherscan.io/accounts/c.

[20] "Exception on overflow #796," https://git.io/vFA8e.

[21] "Formal Verification and Ethereum," https://ethereum.stackexchange.com/questions/11092/what-is-formal-verification-and-why-is-it-important-for-smart-contracts.

[22] "Formal Verification for Solidity Contracts," https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[23] "HackerGold Bug," https://git.io/vFAl2.

[24] "Hyperledger Fabric," https://hyperledger.org/projects/fabric.

[25] "Hyperledger Fabric Mockstub," https://git.io/vFA8Y.

[26] "Inian Parameshwaran," Personal Communication.

[27] "LLVM-based compiler for Go ," https://git.io/FPuO.

[28] "Logical bug," https://live.ether.camp/account/ef71862273817c9e082ca2c92486c8dcdcd9356f/contract.

[29] "Loi Luu," Personal Communication.

[30] "Model arbitrary integer size," https://git.io/vFAlu.

[31] "Multiply your ether," https://etherscan.io/address/0xc357a046c5c13bb4e6d918a208b8b4a0ab2f2efd#code.

[32] "Oyente: An Analysis Tool for Smart Contracts," https://git.io/vFAlX.

[33] "Oyente Results," https://raw.githubusercontent.com/oyente/benchmarks/master/benchmark/results.json.

[34] "Parity MultiSig bug," http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug.

[35] "Remove tx.origin #683," https://git.io/vFA8n.

[36] "Scoping and Declarations," https://solidity.rtfd.io/en/develop/control-structures.html.

[37] "SeaHorn," https://seahorn.github.io/.

[38] "Send w/Throw Is Dangerous," http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful/.

[39] "Simple Dice," https://etherscan.io/address/0x237f29bbFd52C768A02980eA8D4D983a1D234eDC.

[40] "Smack," https://git.io/vFAlB.

[41] "Solidity Programming Language." https://git.io/vFA47.

[42] "StandardToken," https://git.io/vFAlg.

[43] "The DAO," https://en.wikipedia.org/wiki/The_DAO_(organization).

[44] "The DAO is kind of a mess," https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess/.

[45] "Tx.Origin And Ethereum Oh My!" http://vessenes.com/tx-origin-and-ethereum-oh-my/.

[46] "Unchecked-Send Bug," http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[47] "Underhanded Solidity coding," https://redd.it/4e5y30.

[48] "Units and Globally Available Variables," https://solidity.rtfd.io/en/develop/units-and-global-variables.html.

[49] "Wallet," https://etherscan.io/address/0xab7c74abc0c4d48d1bdad5dcb26153fc8780f83e.

[50] "Why3," http://why3.lri.fr/.

[51] "XACML," https://tools.ietf.org/html/rfc7061.

[52] J. Barnat *et al.*, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *CAV '13*.

[53] L. Bauer *et al.*, "Composing Security Policies with Polymer," in *PLDI '05*.

[54] K. Bhargavan *et al.*, "Formal Verification of Smart Contracts: Short Paper," in *PLAS '16*.

[55] N. Bjørner *et al.*, "Program Verification as Satisfiability Modulo Theories," in *SMT '12*.

[56] A. R. Bradely, "SAT-based Model Checking without unrolling," in *VMCAI 2011*.

[57] M. Castro *et al.*, "Practical Byzantine Fault Tolerance," in *OSDI '99*.

[58] K. Delmolino *et al.*, "Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab." in *FC '16*.

[59] U. Erlingsson *et al.*, "IRM Enforcement of Java Stack Inspection," in *S&P '00*.

[60] D. Evans *et al.*, "Flexible Policy-Directed Code Safety," in *S&P '99*.

[61] A. Gurfinkel *et al.*, "The SeaHorn Verification Framework," in *CAV '15*.

[62] A. Juels *et al.*, "The Ring of Gyges: Investigating the Future of Criminal Smart Contracts," in *CCS '16*.

[63] A. Komuravelli *et al.*, "SMT-based Model Checking for Recursive Programs," in *CAV'14*.

[64] A. E. Kosba *et al.*, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," in *S&P '16*.

[65] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*

[66] C. Lattner *et al.*, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO '04*.

[67] L. Luu *et al.*, "Demystifying Incentives in the Consensus Computer," in *CCS '15*.

[68] ——, "Making Smart Contracts Smarter," in *CCS '16*.

[69] K. L. McMillan, "Interpolants and Symbolic Model Checking," in *VMCAI 2007*.

[70] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." [Online]. Available: https://bitcoin.org/bitcoin.pdf

[71] D. Ongaro *et al.*, "In Search of an Understandable Consensus Algorithm," in *USENIX ATC'14*.

[72] Z. Rakamarić *et al.*, "SMACK: Decoupling Source Language Details from Verifier Implementations," in *CAV '14*.

[73] F. Zhang *et al.*, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *CCS '16*.

[74] J. Zhao *et al.*, "Formalizing the LLVM Intermediate Representation for Verified Program Transformations," in *POPL '12*.