SmartCheck: Static Analysis of Ethereum Smart Contracts

Sergei Tikhomirov
University of Luxembourg
Esch-sur-Alzette, Luxembourg
sergey.s.tikhomirov@gmail.com

Ekaterina Voskresenskaya
SmartDec
Moscow, Russia
voskresenskaya@smartdec.net

Ivan Ivanitskiy
SmartDec
Moscow, Russia
ivanitskiy@smartdec.net

Ramil Takhaviev
SmartDec
Moscow, Russia
tahaviev@smartdec.net

Evgeny Marchenko
SmartDec
Moscow, Russia
marchenko@smartdec.net

Yaroslav Alexandrov
SmartDec
Moscow, Russia
alexandrov@smartdec.net

ABSTRACT

Ethereum is a major blockchain-based platform for smart contracts – Turing complete programs that are executed in a decentralized network and usually manipulate digital units of value. Solidity is the most mature high-level smart contract language. Ethereum is a hostile execution environment, where anonymous attackers exploit bugs for immediate financial gain. Developers have a very limited ability to patch deployed contracts. Hackers steal up to tens of millions of dollars from flawed contracts, a well-known example being "The DAO", broken in June 2016. Advice on secure Ethereum programming practices is spread out across blogs, papers, and tutorials. Many sources are outdated due to a rapid pace of development in this field. Automated vulnerability detection tools, which help detect potentially problematic language constructs, are still underdeveloped in this area.

We provide a comprehensive classification of code issues in Solidity and implement SmartCheck – an extensible static analysis tool that detects them[1]. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. We evaluated our tool on a big dataset of real-world contracts and compared the results with manual audit on three contracts. Our tool reflects the current state of knowledge on Solidity vulnerabilities and shows significant improvements over alternatives. SmartCheck has its limitations, as detection of some bugs requires more sophisticated techniques such as taint analysis or even manual audit. We believe though that a static analyzer should be an essential part of contract developers' toolbox, letting them fix simple bugs fast and allocate more effort to complex issues.

KEYWORDS

Ethereum, Solidity, smart contracts, static analysis, bug detection

[1]The source code is available at https://github.com/smartdec/smartcheck.

WETSEB'18, May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5726-5/18/05…$15.00
https://doi.org/10.1145/3194113.3194115

ACM Reference Format:
Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *WETSEB'18: WETSEB'18:IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194113.3194115

1 INTRODUCTION

Ethereum was introduced in 2014 and launched in 2015 [VB+14]. Ethereum nodes store data, execute smart contracts, and maintain a shared view of the global state using a proof-of-work consensus mechanism similar to that in Bitcoin [Tik17]. Contrary to previous attempts at blockchain programming, e.g., Bitcoin scripting, Ethereum language is Turing complete and thus able to express arbitrarily complex logic.

Developers write contracts in high-level languages (the most popular and mature one is Solidity) and compile them to bytecode of the Ethereum virtual machine (EVM) – a stack-based VM operating on 256-bit words[2]. Compared to general purpose VMs like the Java virtual machine, EVM is relatively simple, executes deterministically, and natively supports certain cryptographic primitives [But17]. A contract is deployed by broadcasting a transaction containing its bytecode and initialization parameters. Miners include it in a block, permanently storing the contract at a unique blockchain address. Users interact with the contract by broadcasting transactions with its address, the function to be called, and its arguments. Upon request, the contract can call other contracts and send units of *ether* – the Ethereum native cryptocurrency – to users or other contracts.

To make spamming costly, the Ethereum protocol specifies a cost (denominated in *gas* units) for each EVM operation [Woo14]. A user pays upfront for the expected amount of gas the computation will consume and gets a partial refund after a successful execution. If an exception (including "out of gas") occurs, all state changes are reverted, but the gas may not be refunded[3]. The ether price of a gas unit is determined by the market.

Ethereum allows people and companies globally to programmatically encode and trustlessly enforce complex financial agreements. This opens up new business models and constitutes a dramatic change in the digital economy. New software development tools are required to ensure correctness and security of smart contracts.

[2]It is also possible to write contracts in bytecode directly.
[3]Gas refunds depend on the exception type: `assert` consumes all gas, `require` does not (starting from the Byzantium release in October 2017).

1.1 Security challenges in Ethereum

Security is a primary concern in Ethereum programming for multiple reasons:

- **Unfamiliar execution environment**. Ethereum differs from centrally managed execution environments, be that mobile, desktop, or cloud. Developers are not used to their code being executed by a global network of anonymous, mutually distrusting, profit-driven nodes.
- **New software stack**. The Ethereum stack (the Solidity compiler, the EVM, the consensus layer, etc) is under development, with security vulnerabilities still being discovered [Sol17b].
- **Very limited ability to patch contracts**. A deployed contract can not be patched[4]. This makes a popular "move fast and break things" motto inapplicable: a contract must be correct before deployment.
- **Anonymous financially motivated attackers**. Compared to many cybercrimes, exploiting smart contracts offers higher gains (the prices of cryptocurrencies have been increasing rapidly), easier cashing out (ether and tokens are instantly tradable), and lower risk of punishment due to anonymity.
- **Rapid pace of development**. Blockchain companies strive to release their products fast, often at the expense of security.
- **Suboptimal high-level language**. Some argue that Solidity itself inclines programmers towards unsafe development practices [ydt16].

A textbook example of an Ethereum contract exploit is the DAO hack. The DAO was an Ethereum-based venture capital fund. In May 2016, it collected around $150 million in the largest crowdfunding campaign to date. In June 2016, an unknown hacker exploited multiple vulnerabilities in the DAO code and gained control over ether worth around $50 million at that time [Sir16]. Though the Ethereum protocol executed correctly, the core developers proposed a hard fork to restore stakeholders' deposits, violating the premise of decentralized applications running "exactly as programmed"[5]. More recent examples of high-profile loss of ether due to software vulnerabilities include two incidents with the Parity multi-signature wallet in July and November 2017 [Pal17]. These and many similar events of a smaller scale illustrate the importance of security in Ethereum.

For the purposes of this paper, we assume correctness of the Ethereum core infrastructure and focus on security from a contract developer's viewpoint. We classify issues in Solidity source code and develop a static analysis tool – SmartCheck – that detects them. We test SmartCheck on a large set of real-world contracts and measure the relative prevalence of various code issues. SmartCheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR).

2 CLASSIFICATION OF ISSUES IN SOLIDITY CODE

We classify Solidity code issues as follows (based on [Hen17]):

- **Security** issues lead to exploits by a malicious user account or contract;
- **Functional** issues cause the violation of the intended functionality[6];
- **Operational** issues lead to run-time problems, e.g., bad performance;
- **Developmental** issues make code difficult to understand and improve.

We differentiate between functional and security issues: the former pose problems even without an adversary (though an external malicious actor can aggravate the situation), while the latter do not. Our primary sources are [Con16] [Sol17a] [ABC17] [DAK+15] [CLLZ17] [Sol17c]. See Table 1 for a summary of all issues (the second column denotes severity: 3 – high, 2 – medium, 1 – low).

2.1 Security issues

2.1.1 Balance equality. Avoid checking for strict balance equality: an adversary can forcibly send ether to any account by mining or via `selfdestruct`.

```
if (this.balance == 42 ether) { /* ... */} // bad
if (this.balance >= 42 ether) { /* ... */} // good
```

The pattern detects comparison expressions with == which contain `this.balance` as either left- or right-hand side.

2.1.2 Unchecked external call. Expect calls to external contract to fail. When sending ether, check for the return value and handle errors. The recommended way of doing ether transfers is `transfer` (see Section 2.1.4).

```
addr.send(42 ether); // bad
if (!addr.send(42 ether)) revert; // better
addr.transfer(42 ether); // good
```

The pattern detects an external function call (`call`, `delegatecall`, or `send`) which is not inside an `if`-statement.

2.1.3 DoS by external contract. A conditional statement (`if`, `for`, `while`) should not depend on an external call: the callee may permanently fail (`throw` or `revert`), preventing the caller from completing the execution.

In the following example, the caller expects the oracle to return an integer value (`badOracle.answer()`), but the actual oracle implementation may throw an exception in some or all cases.

```
function dos(address oracleAddr) public {
  badOracle = Oracle(oracleAddr);
  if (badOracle.answer() < 42) { revert; }
  // ...
}
```

This rule contains multiple patterns:

- an `if`-statement with an external function call in the condition and a `throw` or a `revert` in the body;
- a `for`- or an `if`-statement with an external function call in the condition.

[4]Though workarounds exist, such as proxy contracts redirecting calls to an adaptable address of the latest version of the main contract.

[5]Concerns about Ethereum's governance lead to the creation of Ethereum Classic [Eth17b] – a continuation of the Ethereum blockchain without the DAO fork.

[6]Though without a specification we only assume what the intended functionality is.

2.1.4 `send` *instead of* `transfer`. The recommended way to perform ether payments is `addr.transfer(x)`, which automatically throws an exception if the transfer is unsuccessful, preventing the problem described in Section 2.1.2. The pattern detects the `send` keyword.

2.1.5 Re-entrancy. Consider the following code:

```
pragma solidity 0.4.19;
contract Fund {
  mapping(address => uint) balances;
  function withdraw() public {
    if (msg.sender.call.value(balances[msg.sender])())
    balances[msg.sender] = 0;
  }
}
```

The contract at `msg.sender` can get multiple refunds and retrieve all `Fund`'s ether by recursively calling `withdraw` before its share is set to `0`. Besides, it can modify the state of some third contract, which `Fund` depends on. Use the "checks – effects – interactions" pattern: first check the invariants, then update the internal state, then communicate with external entities (see also Section 2.1.4):

```
function withdraw() public {
  uint balance = balances[msg.sender];
  balances[msg.sender] = 0;
  msg.sender.transfer(balance);
  // state reverted, balance restored if transfer fails
}
```

The pattern detects an external function call which is followed by an internal function call.

2.1.6 Malicious libraries. Third-party libraries can be malicious. Avoid external dependencies or ensure that third-party code implements only the intended functionality. The pattern simply detects the `library` keyword (and thus produces some false positives).

2.1.7 *Using* `tx.origin`. Contracts can call each others' public functions. `tx.origin` is the first account in the call chain (always an externally owned one, i.e., not a contract); `msg.sender` is the immediate caller. For instance, in a call chain A → B → C, from the C's viewpoint, `tx.origin` is A, and `msg.sender` is B.

Use `msg.sender` instead of `tx.origin` for authentication. Consider a wallet:

```
pragma solidity 0.4.19;
contract TxWallet {
    address private owner;
    function TxWallet() { owner = msg.sender; }
    function transferTo(address dest, uint amount) public
          {
        require(tx.origin == owner);  // authentication
        dest.transfer(amount);
    }
}
```

User sends ether to the address of the `TxAttackerWallet`, which forwards the call to a `TxWallet` and obtains all funds, acting as the user (`tx.origin`):

```
pragma solidity 0.4.19;
interface TxWallet {
    function transferTo(address dest, uint amount);
}
contract TxAttackerWallet {
    address private owner;
    function TxAttackerWallet() { owner = msg.sender; }
    function() payable {
        TxWallet(msg.sender).transferTo(owner, msg.sender
            .balance);
    }
}
```

The pattern detects the environmental variable `tx.origin`.

2.1.8 Transfer forwards all gas. Solidity provides many ways to transfer ether (see Section 2.1.4). `addr.call.value(x)()` transfers `x` ether and forwards all gas to `addr`, potentially leading to vulnerabilities like re-entrancy (see Section 2.1.5). The recommended way to transfer ether is `addr.transfer(x)`, which only provides the callee with a "stipend" of 2300 gas. The pattern detects functions whose name is `call.value` and whose argument list is empty.

2.2 Functional issues

2.2.1 Integer division. Solidity supports neither floating-point nor decimal types. For integer division, the quotient is rounded down. Account for it, especially when calculating ether or token amounts. The pattern detects division (`/`) where the the numerator and the denominator are number literals.

2.2.2 Locked money. Contracts programmed to receive ether should implement a way to withdraw it, i.e., call `transfer` (recommended), `send`, or `call.value` at least once. The patterns detects contracts that contain a `payable` function but contain neither of the withdraw-enabling functions mentioned above.

2.2.3 Unchecked math. Solidity is prone to integer over- and underflow[7]. Overflow leads to unexpected effects and can lead to loss of funds if exploited by a malicious account. Use the SafeMath library[8] that checks for overflows (multiple implementations exist, e.g. [Saf17]). The pattern detects arithmetic operations `+`, `-`, `*`, which are not inside a conditional statement. This rule was temporarily muted for testing (Section 4) due to a high false positive rate.

2.2.4 Timestamp dependence. Miners can manipulate environmental variables and are likely to do so if they can profit from it. Consider a lottery that distributes prizes depending on whether `now` (alias for `block.timestamp`) is odd or even:

```
  if (now % 2 == 0) winner = pl1; else winner = pl2;
```

A miner can tweak the timestamp and gain unfair advantage. Use block numbers and average time between blocks to estimate the current time. Use secure sources of randomness, such as RANDAO [Ran17]. The pattern detects the environmental variable `now`.

2.2.5 Unsafe type inference. Solidity supports type inference: the type of `i` in `var i = 42;` is the smallest integer type sufficient to store the right-hand side value (`uint8`). Consider a `for`-loop:

```
for (var i = 0; i < array.length; i++) { /*...*/ }
```

The type of `i` is inferred to `uint8`. If `array.length` is bigger than 256, an overflow will occur. Explicitly define the type when declaring integer variables:

```
for (uint256 i = 0; i < array.length; i++) { /*...*/ }
```

The pattern detects assignments where the left-hand side is a `var` and the right-hand side is an integer (matches `^[0-9]+$`).

[7] Referred to as simply overflow for brevity.
[8] See Section 2.1.6 for advice on library usage.

2.3 Operational issues

2.3.1 Byte array. Use `bytes` instead of `byte[]` for lower gas consumption. The pattern detects the construction `byte[]`.

2.3.2 Costly loop. Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized cloud providers. Moreover, Ethereum miners impose a limit on the total number of gas consumed in a block. In the following example, if `array.length` is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed:

```
for (uint256 i = 0; i < array.length; i++) { costlyF(); }
```

This becomes a security issue, if an external actor influences `array.length`. E.g., if `array` enumerates all registered addresses, and registration is open, an adversary can register many addresses, causing denial of service. The rule includes two patterns:

- a `for`-statement with a function call or an identifier inside the condition;
- a `while`-statement with a function call inside the condition.

2.4 Developmental issues

2.4.1 Token API violation. ERC20 is the de-facto standard API for implementing *tokens* – transferable units of value managed by a contract. Exchanges and other third-party services may struggle to integrate a token that does not conform to it. Certain ERC20 functions (`approve`, `transfer`, `transferFrom`) return a `bool` indicating whether the operation succeded. It is not recommended to throw exceptions (`revert`, `throw`, `require`, `assert`) inside those functions. Note that library functions may also throw exceptions (see Section 2.1.6).

```
function transferFrom(address _spender, uint _value)
returns (bool success) {
  require (_value < 20 wei);
  // ...
}
```

The pattern detects a contract inherited from a contract with a name including the word "token", which may throw exceptions from inside one of the functions mentioned above.

2.4.2 Compiler version not fixed. Solidity source files indicate the versions of the compiler they can be compiled with:

```
pragma solidity ^0.4.19;  // bad: 0.4.19 and above
pragma solidity 0.4.19;   // good: 0.4.19 only
```

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee. The pattern detects the version operator ^ in the `pragma` directive.

2.4.3 `private` modifier. Contrary to a popular misconception, the `private` modifier does not make a variable invisible. Miners have access to all contracts' code and data. Developers must account for the lack of privacy in Ethereum. The pattern detects state variable declarations with a `private` modifier.

2.4.4 Redundant fallback function. Contracts should reject unexpected payments (see Sections 2.1.1, 2.2.2). Before Solidity 0.4.0, it was done manually:

```
function () payable { throw; }
```

Starting from Solidity 0.4.0, contracts without a fallback function automatically revert payments, making the code above redundant. The pattern detects the described construction (only if the `pragma` directive indicates the compiler version not lower than 0.4.0).

2.4.5 Style guide violation. In Solidity, function[9] and event names usually start with a lower- and uppercase letter respectively:

```
function Foo(); // bad
event logFoo(); // bad
function foo(); // good
event LogFoo(); // good
```

Violating the style guide decreases readability and leads to confusion. The pattern detects the described constructions.

2.4.6 Implicit visibility level. The default function visibility level in Solidity is `public`. Explicitly define function visibility to prevent confusion.

```
function foo() { /*...*/ } // bad
function foo() public { /*...*/ } // good
function bar() private { /*...*/ } // good
```

The pattern detects function and variable definitions without a visibility modifier.

3 AUTOMATED ANALYSIS OF SMART CONTRACTS

3.1 Approaches to code analysis

Dynamic code analysis runs the program and considers only a subset of all execution paths on some input data [LSCL12]. Static code analysis may guarantee full coverage without executing the program and may run fast enough on code of reasonable size. Static analysis usually includes three stages:

(1) building an intermediate representation (IR), such as abstract syntax tree or three-address code, for a deeper analysis compared to analyzing text;
(2) enriching the IR with additional information [Wög05] using algorithms such as control- and dataflow analysis (synonym, constant, and type propagation [ASU07]), taint analysis [TPF+09], symbolic execution, abstract interpretation;
(3) vulnerability detection w.r.t. a database of patterns, which define vulnerability criteria in IR terms.

In this paper, we do not consider formal verification methods, as they require a rarely available formal specification of the contract's intended functionality.

3.2 SmartCheck

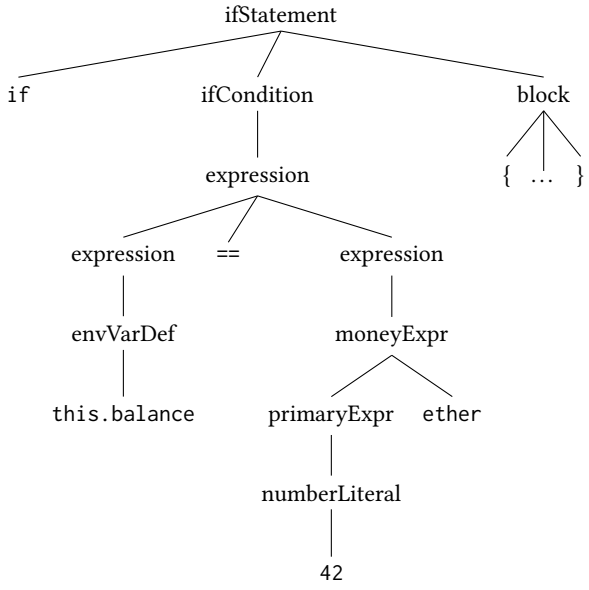
We propose SmartCheck – a static analysis tool for Ethereum smart contracts implemented in Java. SmartCheck runs lexical and syntactical analysis on Solidity source code. It uses ANTLR [ant17] and a custom Solidity grammar to generate an XML parse tree [ASU07] as an intermediate representation (IR). We detected vulnerability patterns by using XPath [xpa] queries on the IR. Thus SmartCheck provides full coverage: the analyzed code is fully translated to the IR, and all its elements can be reached with XPath matching. Line numbers are stored as XML attributes and help localize findings in

[9] With the exception of constructors: they must share the name with the contract and thus usually start with an uppercase letter.

Table 1: Code issues detected by SmartCheck (gray background – false positives possible)

Name	S.	Description
Balance equality (2.1.1)	2	Adversary can manipulate contract logic by forcibly sending it ether. Use non-strict inequality on balances
Unchecked external call (2.1.2)	3	The return value is not checked. Always check return values of functions
DoS by external contract (2.1.3)	3	Expect external calls to deliberately `throw`
`send` instead of `transfer` (2.1.4)	2	The return value of `send` should be checked. Use `transfer`, which is equivalent to `if (!send()) throw;`
Re-entrancy (2.1.5)	3	External contracts should be called after all local state updates
Malicious libraries (2.1.6)	1	Using external libraries may be dangerous. Avoid external code dependencies, audit all code that is part of the project
Using `tx.origin` (2.1.7)	2	A malicious contract can act on a user's behalf. Use `msg.sender` for authentication
Transfer forwards all gas (2.1.8)	3	`a.call.value()()` forwards all gas, allowing the callee to call back. Use `a.transfer()`: it only provides the callee with 2300 gas (insufficient for a callback)
Integer division (2.2.1)	1	The quotient is rounded down. Account for it, especially for ether and token amounts
Locked money (2.2.2)	2	The contract receives ether, but there is no way to withdraw it. Implement a withdraw function or reject payments
Unchecked math (2.2.3)	1	Without extra checks, integer over- and underflow is possible. Use SafeMath
Timestamp dependence (2.2.4)	2	Miners can alter timestamps. Make critical code independent of the environment
Unsafe type inference (2.2.5)	2	Type inference choses the smallest integer type possible. Explicitly specify types
Byte array (2.3.1)	1	`byte[]` requires more than `bytes`
Costly loop (2.3.2)	2	Expensive computation inside loops may exceed the block gas limit. Avoid loops with big or unknown number of steps
Token API violation (2.4.1)	1	The contract `throws` where the ERC20 standard expects a `bool`. Return `false` instead
Compiler version not fixed (2.4.2)	1	Contract compiles with future compiler versions. Specify the exact compiler version
`private` modifier (2.4.3)	1	The `private` modifier does not hide the variable's value, only prevents external contracts from editing it
Redundant fallback function (2.4.4)	1	The payment rejection fallback is redundant. Remove the function to save space: payments are rejected automatically
Style guide violation (2.4.5)	1	Unfamiliar capitalization style causes confusion. Start function names with lowercase, events with uppercase
Implicit visibility level (2.4.6)	1	Functions are `public` by default. Avoid ambiguity: explicitly declare visibility level

Figure 1: Parse tree for the Balance equality code example



source code. IR attributes can be enriched with additional information when new analysis methods are implemented. The tool can be extended to support other smart contact languages by adding an ANTLR grammar and a pattern database (IR-level algorithms remain unchanged).

As an example, consider the Balance equality issue (2.1.1). We aim to detect constructions that test the contract's balance for equality, for instance:

```
if (this.balance == 42 ether){...}.
```

The parse tree of this construction is shown in Figure 1.

The corresponding XPath pattern is shown in Listing 1.

```
//expression[expression//envVarDef
[matches(text()[1],"^this.balance$")]]
[matches(text()[1],"^==|!=$")]
```

Listing 1: XPath pattern for the Balance equality issue

In this case we do not expect false positives, as we are able to precisely describe the target construction in XPath[10].

More complex rules can not be precisely described with XPath, which leads to false positives. Consider the Re-entrancy issue (2.1.5). SmartCheck reports violations of the Checks-Effects-Interactions (CEI) pattern, which does not always lead to re-entrancy (Listing 2).

```
pragma solidity 0.4.19;
contract Foo {
  bool inBar = false;
  function bar(address someAddress) {
    if (inBar) throw;
    inBar = true;
    someAddress.transfer(0);
    inBar = false;
  }
}
```

Listing 2: Violation of CEI not leading to re-entrancy

[10] Assuming that ANTLR builds the AST correctly based on the Solidity grammar.

4 EXPERIMENTAL RESULTS

4.1 Goals and definitions

We compare SmartCheck with three freely available tools aimed at statically detecting vulnerabilities in Ethereum contracts: Oyente, Remix, and Securify[11] and with the results of manual audit.

We define a true finding as an issue (detected by a tool with manual verification or manually) that is a bad practice and should be fixed from our viewpoint (it may or may not be an exploitable vulnerability). All issues found by the tools were manually labeled as either true positive (TP) or false positive (FP). A false negative (FN) for each of the four tools (Oyente, Remix, Securify, and SmartCheck) is a true finding that was not detected by this tool.

For each tool, the false discovery rate (FDR) is the number of FPs for this tool divided by the number of all issues reported by this tool: FDR = FP / (TP + FP). False negative rate (FNR) is the number of FNs for this tool divided by the number of all true findings (found by any of the tools or manually), which is the sum of TP and FN for this tool: FNR = FN / (TP + FN).

Section 4.2 assesses FDR and FNR of SmartCheck and three other tools on three typical contracts. Section 4.3 measures the prevalence of code issues on a large set of real-world contracts.

4.2 Case studies

We consider three contracts: Genesis ("the platform for the private trust management market" [Gen17a], source code [Gen17b], analyzed at commit `1ecf99d`), Hive ("the first crypto currency invoice financing platform" [Hiv17a], source code [Hiv17b], analyzed at commit `0d54699`), and Populous ("an online platform that matchmakes invoice sellers to invoice buyers hosted on the blockchain" [Pop17a], source code [Pop17b], analyzed at commit `10de4ae`).

The FDR and FNR for each tool (in % and in absolute numbers) are presented in Table 2.

Oyente and Securify did not show any TPs on these threee contracts. Remix detected TPs only in the Populous contract. Remix and SmartCheck showed an overall FDR of 97% and 69% respectively, and an overall FNR of 92% and 47% respectively. This means that SmartCheck showed better FDR and FNR compared to its closest competitor. Overall, SmartCheck reported 87 issues in the three contracts.

Requirements for code analysis tools differ across platforms and domains. Due to a peculiar security landscape in smart contract programming (see Section 1.1), low FN rate is crucial (a missed vulnerability can be disastrous), whereas a relatively high FP rate is tolerable: most contracts contain only a few hundreds of lines of code (see Section 4.3) and can be audited manually.

Though SmartCheck's FDR of 69% may seem pretty high, it is not a serious issue in this domain. 47% is a reasonable level of FNR, since many vulnerabilities in smart contracts are related to business logic and can not be detected automatically. Most of SmartCheck's FNs were found manually (not by other tools).

SmartCheck detected a critical issue in one of the contracts: an attacker could create an unlimited number of internal entities and block the normal operation of the contract. A public function (i.e., such that any Ethereum user can call it) allowed to add an element to an internal array (Listing 3). Several critical functions then iterated through this array (e.g., Listing 4), so an attacker could make those functions permanently fail (a function call would require more gas than the block gas limit).

```
function createGroup(string _name, uint _goal)
    onlyOpenAuction
    returns (uint8 err, uint groupIndex)
{
  if(checkDeadline() == false && _goal >= fundingGoal &&
       _goal <= invoiceAmount) {
    groupIndex = groups.length++;
    groups[groupIndex].groupIndex = groupIndex;
    groups[groupIndex].name = _name;
    groups[groupIndex].goal = _goal;

    EventGroupCreated(groupIndex, _name, _goal);

    return (0, groupIndex);
  } else {
    return (1, 0);
  }
}
```

Listing 3: Adding an element to the internal array

```
function findBidder(bytes32 bidderId) constant returns (
    uint8 err, uint groupIndex, uint bidderIndex) {
  for(groupIndex = 0; groupIndex < groups.length;
      groupIndex++) {
    for(bidderIndex = 0; bidderIndex < groups[groupIndex
        ].bidders.length; bidderIndex++) {
      if (Utils.equal(groups[groupIndex].bidders[
          bidderIndex].bidderId, bidderId) == true) {
        return (0, groupIndex, bidderIndex);
      }
    }
  }
  return (1, 0, 0);
}
```

Listing 4: Iterating through the internal array

4.3 Testing on a massive sample

We downloaded the source code of 4,600 verified contracts (1,537,954 lines of code) as of 4 October 2017 from Etherscan [Eth17a] using a Java library JSoup [JSo17] and ran SmartCheck on this dataset.

The contract balances differ significantly (see Figure 2). The vast majority (3984, or 86.6%) of contracts have a zero balance. One contract holds over one million ether (1,500,000, or $440 million at the time of testing), which accounts for 38.4% of the total balance of all contracts. Contracts have from 1 to 2,525 lines of code, with an average of 334 lines and a median of 221 lines.

SmartCheck analyzed the dataset in 7644 seconds (437 lines per second[12]). As per SmartCheck, 99.9% of contracts have issues, 63.2% of contracts have critical vulnerabilities[13]. The findings are presented in Table 3 and Figure 3 (colors denote severity levels: black – high, dark gray – medium, light gray – low). The most prevalent issue, **Implicit visibility level** (detected 81160 times,

[11] In case of Securify, we consider only partial results, since full results are not displayed in the publicly available version of the tool.

[12] Intel Core i5-4210M @ 2.60GHz, 12 GB RAM, Windows 8.1 64 bit

[13] The issues found by SmartCheck in the big dataset were not manually verified.

Table 2: Tools results on the three projects and overall

Project		Oyente	Remix	Securify	SmartCheck
Genesis Vision	TP/FP/FN	0/6/10	0/40/10	0/19/10	7/22/3
	FDR (%)	100	100	100	75.86
	FNR (%)	100	100	100	30.00
Hive	TP/FP/FN	0/6/22	0/11/22	0/6/22	6/7/16
	FDR (%)	100	100	100	53.85
	FNR (%)	100	100	100	72.73
Populous	TP/FP/FN	0/7/19	4/60/15	0/45/19	14/31/5
	FDR (%)	100	93.75	100	68.89
	FNR (%)	100	78.95	100	26.32
Overall	TP/FP/FN	0/19/51	4/111/47	0/70/51	27/60/24
	FDR (%)	100	96.52	100	68.97
	FNR (%)	100	92.16	100	47.06

Figure 2: Distribution of non-zero contract balances (ether)

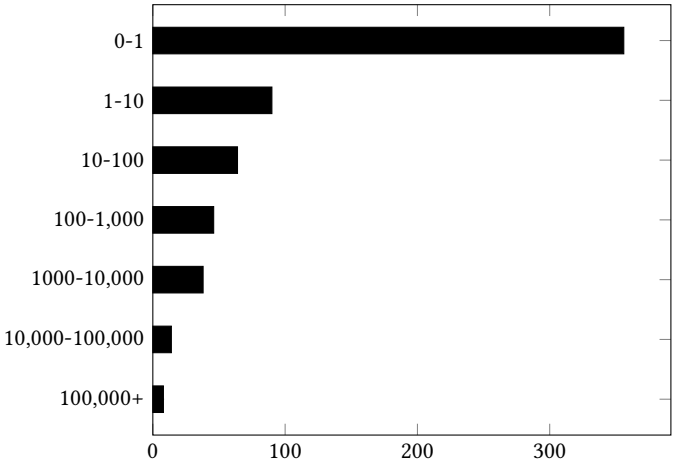
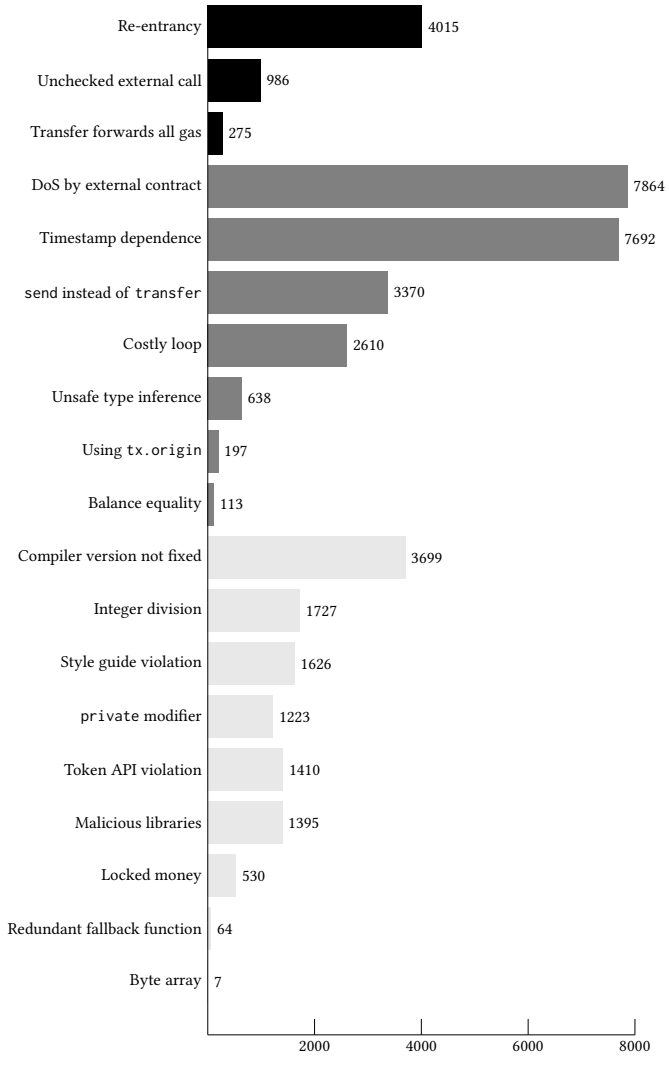


Figure 3: Findings on the big dataset (excluding Implicit visibility level)



which accounts for 67.296% of all findings), is excluded from the figure for clarity.

5 RELATED WORK

Multiple tools aim at improving the security and correctness of Ethereum smart contracts. Static checks are built into the online Solidity compiler Remix [Rem17]. Oyente [LCO+18] is a symbolic execution tool vulnerability detection in EVM bytecode. Securify [Sec17] analyzes Solidity source code as well as EVM bytecode. [BDLF+16] and [PE16] propose writing Ethereum contracts in safer languages (F* and Idris respectively). [Hir17b] describes existing attempts to formal verification of EVM bytecode as well as the EVM itself. [Hir17a] and [LCO+16] use symbolic execution to analyze EVM bytecode. [HSZ+17] formally describes the full semantics of the EVM, providing the foundation for formal verification tools for EVM bytecode .

Table 3: Code issues detected on a big dataset

Severity	Pattern	Findings	% of all
high	Re-entrancy	4015	3.329
	Unchecked external call	986	0.818
	Transfer forwards all gas	275	0.228
medium	DoS by external contract	7864	6.521
	Timestamp dependence	7692	6.378
	`send` instead of `transfer`	3370	2.794
	Costly loop	2610	2.164
	Unsafe type inference	638	0.529
	Using `tx.origin`	197	0.163
	Balance equality	113	0.094
low	Implicit visibility level	81160	67.296
	Compiler version not fixed	3699	3.067
	Integer division	1727	1.432
	Style guide violation	1626	1.348
	`private` modifier	1223	1.014
	Token API violation	1410	1.169
	Malicious libraries	1395	1.157
	Locked money	530	0.439
	Redundant fallback function	64	0.053
	Byte array	7	0.006

6 CONCLUSION AND FUTURE WORK

We provided a comprehensive overview and classification of the currently known code issues in Solidity – the major high-level language for Ethereum smart contracts. We implemented SmartCheck – an efficient static analysis tool for Solidity, which offers significant improvements over existing alternatives. We tested out tool on a massive set of real-world contracts and detected code issues in the vast majority of them.

The tool can be improved in multiple directions: improving the grammar[14], making patterns more precise (e.g., the temporarily muted Unchecked math), adding new patterns, implementing more sophisticated static analysis methods, adding support for other languages.

Security is still an issue in blockchain development. We hope that SmartCheck will help solve this major challenge by providing smart contract developers with fast and relevant feedback on potentially problematic source code patterns.

REFERENCES

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *POST*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017. http://eprint.iacr.org/2016/1007.

[ant17] ANTLR, 2017. http://www.antlr.org/.

[ASU07] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-Wesley Reading, 2007.

[BDLF+16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[But17] Vitalik Buterin. Design rationale, 2017. https://github.com/ethereum/wiki/wiki/Design-Rationale.

[CLLZ17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *SANER*, pages 442–446. IEEE Computer Society, 2017.

[Con16] Ethereum contract security techniques and tips, 2016. https://github.com/ConsenSys/smart-contract-best-practices.

[DAK+15] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, , and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Cryptology ePrint Archive, Report 2015/460, 2015. http://eprint.iacr.org/2015/460.

[Eth17a] Contracts with verified source codes only, 2017. https://etherscan.io/contractsVerified.

[Eth17b] Ethereum Classic, 2017. https://ethereumclassic.github.io/.

[Gen17a] Genesis, 2017. https://genesis.vision/.

[Gen17b] Genesis Github repository, 2017. https://github.com/GenesisVision/ico-contracts/.

[Hen17] Kevlin Henney. Inside requirements, 2017. https://www.slideshare.net/Kevlin/inside-requirements.

[Hir17a] Yoichi Hirai. Dr. Y's Ethereum contract analyzer, 2017. http://dry.yoichihirai.com/.

[Hir17b] Yoichi Hirai. Formal verification of Ethereum contracts, 2017. https://github.com/pirapira/ethereum-formal-verification-overview.

[Hiv17a] Hive, 2017. https://bitcointalk.org/index.php?topic=1959159.0.

[Hiv17b] Hive Github repository, 2017. https://github.com/HiveProjectLtd/HVNTokenBasic/.

[HSZ+17] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. KEVM: A complete semantics of the Ethereum virtual machine. 2017. https://hdl.handle.net/2142/97207.

[JSo17] jsoup: Java HTML parser, 2017. https://jsoup.org/.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. http://eprint.iacr.org/2016/633.

[LCO+18] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Oyente, 2018. http://www.comp.nus.edu.sg/~loiluu/oyente.html.

[LSCL12] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. Software vulnerability discovery techniques: a survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security (MINES)*, pages 152–156. IEEE, 2012.

[Pal17] Santiago Palladino. The Parity wallet hack reloaded, 2017. https://blog.zeppelin.solutions/the-parity-wallet-hack-reloaded-91bbfa5e510c.

[PE16] Jack Pettersson and Robert Edström. Safer smart contracts through type-driven development. 2016. https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf.

[Pop17a] Populous, 2017. http://populous.co/.

[Pop17b] Populous Github repository, 2017. https://github.com/bitpopulous/Populous-smart-contracts/.

[Ran17] RANDAO: a DAO working as RNG of Ethereum, 2017. https://github.com/randao/randao.

[Rem17] Remix - Solidity IDE, 2017. https://ethereum.github.io/browser-solidity/.

[Saf17] Safemath, 2017. https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.sol.

[Sec17] Securify. Formal verification of Ethereum smart contracts, 2017. http://securify.ch/.

[Sir16] Emin Gün Sirer. Thoughts on The DAO hack, 2016. http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/.

[Sol17a] Solidity official documentation, 2017. https://solidity.readthedocs.io/.

[Sol17b] Solidity version 0.4.14, 2017. https://github.com/ethereum/solidity/releases/tag/v0.4.14.

[Sol17c] Zeppellin Solutions, 2017. https://blog.zeppelin.solutions/tagged/security.

[Tik17] Sergei Tikhomirov. Ethereum: state of knowledge and research perspectives. 2017. https://hdl.handle.net/10993/32468.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.

[VB+14] Fabian Vogelsteller, Vitalik Buterin, et al. Ethereum whitepaper, 2014. https://github.com/ethereum/wiki/wiki/White-Paper.

[Wög05] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, Tech. rep., Technische Universität Wien, 2005.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014. http://yellowpaper.io/.

[xpa] XPath. https://www.w3.org/TR/xpath20/.

[ydt16] ydtm. The bug which the DAO hacker exploited was not merely in the DAO itself, 2016. https://redd.it/4opjov.

[14]The currently used grammar failed to parse 0.16% of lines in our dataset.