Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts

Christof Ferreira Torres
SnT, University of Luxembourg
Luxembourg, Luxembourg
christof.torres@uni.lu

Julian Schütte
Fraunhofer AISEC
Garching, Germany
julian.schuette@aisec.fraunhofer.de

Radu State
SnT, University of Luxembourg
Luxembourg, Luxembourg
radu.state@uni.lu

ABSTRACT

The capability of executing so-called *smart contracts* in a decentralised manner is one of the compelling features of modern blockchains. Smart contracts are fully fledged programs which cannot be changed once deployed to the blockchain. They typically implement the business logic of distributed apps and carry billions of dollars worth of coins. In that respect, it is imperative that smart contracts are correct and have no vulnerabilities or bugs. However, research has identified different classes of vulnerabilities in smart contracts, some of which led to prominent multi-million dollar fraud cases. In this paper we focus on vulnerabilities related to integer bugs, a class of bugs that is particularly difficult to avoid due to some characteristics of the Ethereum Virtual Machine and the Solidity programming language.

In this paper we introduce Osiris – a framework that combines symbolic execution and taint analysis, in order to accurately find integer bugs in Ethereum smart contracts. Osiris detects a greater range of bugs than existing tools, while providing a better specificity of its detection. We have evaluated its performance on a large experimental dataset containing more than 1.2 million smart contracts. We found that 42,108 contracts contain integer bugs. Besides being able to identify several vulnerabilities that have been reported in the past few months, we were also able to identify a yet unknown critical vulnerability in a couple of smart contracts that are currently deployed on the Ethereum blockchain.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**; *Software security engineering*; Logic and verification;

KEYWORDS

Ethereum, smart contracts, integer bugs, taint analysis, symbolic execution.

ACM Reference Format:
Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274737

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '18, December 3–7, 2018, San Juan, PR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274737

1 INTRODUCTION

Since the release of Satoshi Nakamoto's Bitcoin in 2009, a diverse range of blockchain implementations has emerged. All of these blockchain-based technologies pursue a common goal, namely decentralising the control of a particular asset. They achieve this by substituting trusted central entities with a large network of untrusted entities who strive to reach consensus on a correct history of transactions. Trust is obtained via the assumption that the majority of these nodes act faithfully and respect the blockchain protocol, in order to secure the operation of the blockchain as a whole. Bitcoin's asset is its cryptocurrency, and the trusted centralised entities it attempts to replace, are traditional banks. Modern blockchains such as Ethereum go a step further. The latter aims to decentralise the computer as a whole through the Ethereum Virtual Machine (EVM) [38]. The EVM empowers the distributed execution of programs, in the form of so-called *smart contracts*, deployed on the Ethereum network. The EVM is a purely stack-based virtual machine that supports an instruction set of 134 opcodes in order to be able to execute Turing-complete programs. Smart contract functions are invoked via transactions. Each operation on the EVM costs a certain amount of *gas*. When the total amount of gas assigned to a transaction is exceeded, program execution is terminated and its effects are rolled back. As the gas price is coupled to ether, developers are motivated to write efficient programs to keep transaction costs low and to avoid infinite loops on the EVM.

Developers usually write smart contract code in a high-level language which compiles into EVM bytecode. Although various experimental version of high-level languages exist, at the time of this writing, Solidity [32] is the most prevalent language for developing smart contracts. At a first glance, the C/JavaScript-like syntax of Solidity looks familiar to developers with experience in JavaScript or C, and encourages rapid development of smart contracts. However, the way how smart contracts are executed, as well as their security properties, are fundamentally different from traditional programs and may lead to unexpected behaviour at runtime. In combination with a lack of strict static validation and limited support of development tools, developers are encouraged to tweak their smart contract code until it "just works". While this might be a feasible approach for prototyping, it likely results in fatal errors when smart contracts are deployed as real-world decentralised applications (DApps), on the public Ethereum blockchain. In contrast to traditional programs, once smart contracts have been committed to the blockchain, they cannot be updated anymore. Transactions that were never intended by the developer are irreversible.

With the DAO hack in June 2016 [31], it became obvious what consequences emerge when subtle programming mistakes, in non-updatable smart contracts, hit high-volume DApps. An attacker managed to drain $60 million worth of *ether* (Ethereum cryptocurrency) from the DAO, exploiting a *"re-entrancy"* bug in conjunction with a *"call to the unknown"* bug, both contained in the smart contract. Interestingly, Atzei et al. reviewed the DAO hack one year later and realised that the attack could have been exploited more efficiently, using only two calls to the `fallback` function. The attack makes use of the previous reported vulnerabilities and a new unreported vulnerability: an integer underflow in the `withdraw` function at line 10 (see Listing 9 in Appendix B). The attack works as follows: the attacker first deploys `Mallory2`, then invokes `attack` to donate 1 wei (10^{-18} ether) to herself and subsequently withdraws it (see Listing 10 in Appendix B). The function `withdraw` checks whether the user has enough credit and if so transfers the ether to `Mallory2`. As in the original attack, `call` invokes `Mallory2`'s fallback, which in turn calls back `withdraw`, which is interrupted before updating the credit: hence, the check at line 8 succeeding again. Consequently, the DAO sends 1 wei to `Mallory2` for the second time and invokes her fallback again. However this time the fallback does nothing and the nested calls begin to close. The effect is that `Mallory2`'s credit is updated twice: the first time to zero and the second time to $2^{256} - 1$ wei, because of the underflow. To finalise the attack, the attacker simply invokes `getJackpot`, which transfers all the ether from the DAO to `Mallory2`. The DAO hack did not remain the only attack on the Ethereum blockchain. Since then, a number of other attacks have followed, such as the Parity multi-signature wallet attack, which allowed attackers to steal more than $150 million dollars worth of ether [28]. Moreover, recently, a variety of attacks on Ethereum tokens have been reported, each exploiting an integer overflow [17].

In response to these attacks, academia proposed numerous different solutions to check smart contracts for vulnerabilities, prior to deploying them on the blockchain. These mainly include attempts on formal verification [1, 2, 12, 15, 16, 30] and symbolic execution [18, 20, 23, 25, 26, 33]. While symbolic execution has shown to be a promising approach to identify vulnerabilities in EVM bytecode, so far, only few vulnerability classes have been covered by existing tools. Especially the way how Solidity and the EVM handle integer types may lead to unexpected border cases, introducing potential vulnerabilities in smart contracts.

In this paper, we investigate the prevalence of integer bugs in smart contracts. We introduce Osiris, a symbolic-execution tool for detecting various types of integer bugs in Ethereum smart contracts. We use Osiris to find vulnerabilities in smart contracts deployed to the current Ethereum blockchain. Furthermore, we investigate whether this type of vulnerability is already actively exploited, and point out improvements to the EVM and the Solidity compiler as a safeguard against these types of bugs. In summary, with this paper we contribute in the following aspects:

- We present Osiris, a symbolic execution tool which automatically detects integer bugs in EVM bytecode. The tool currently covers three different types of integer bugs: *arithmetic bugs*, *truncation bugs* and *signedness bugs*.
- We run Osiris on all smart contracts that have been deployed on Ethereum until January 2018, and find that 42,108 of them suffer from at least on these three bugs.
- We compare Osiris to Zeus [18]. Osiris detects more vulnerabilities, with more confidence, as the false positives rate is considerably lower with our tool.
- We analyse 495 Ethereum tokens and discovered vulnerabilities in a couple of them, unknown to the best of our knowledge.
- We propose modifications to the EVM and Solidity compiler, to safeguard against integer bugs.

The remainder of this paper is organised as follows. In the next section, we provide the necessary background knowledge to understand Ethereum smart contracts, and how to detect integer bugs contained in these contracts. Section 3 presents our methodology, and provides an overview on the techniques we used as well as the challenges we had to address. Section 4 presents our tool Osiris and describes its implementation details. Section 5 presents the results of our experiments. In Section 6 we provide a discussion on how to make the Ethereum blockchain safer by suggesting improvements to the Solidity compiler and the EVM. Section 7 summarises previous related work. Finally, in Section 8 we offer our conclusions and future work.

2 BACKGROUND

2.1 The Ethereum Virtual Machine

Smart contracts are executed by miners using the Ethereum Virtual Machine (EVM). The EVM is a stack-based, register-less virtual machine running EVM bytecode represented by an instruction set of 134 8-bit opcodes, at the time of writing. The effect of executing a smart contract is a manipulation of the overall *world state* σ, which is a structure holding the account state of all 160-bit account addresses. An *account state* is comprised of the number of transactions sent from the account address, its balance in Wei, the Merkle trie hash of the account's storage and a hash of the account's bytecode (if the account is a smart contract).

2.1.1 Bytecode Execution. Although the instruction set allows Turing-complete programs, the number of instructions executed in a smart contract is limited so that EVM is typically referred to as a "quasi-Turing" complete language. The purpose of this limitation is to ensure termination of every smart contract at some point, as otherwise miners would be stuck in endless loops and verification of the blockchain would become impossible. The limitation of instructions is achieved by the concept of *gas*, which introduces costs for the execution of every single instruction. When issuing a transaction (i.e. calling a smart contract, for example), the sender has to specify gas limit and gas price. The gas limit is given in units of gas and must be large enough to cover the amount of gas consumed by the instructions of the called contract (plus an overhead for input data), otherwise execution will terminate abnormally with an `OutOfGasException` and its effects will be reverted. During execution of a contract, the EVM holds a *machine state* $\mu = (g, pc, m, i, s)$ where g is gas available, pc is program counter, m is memory contents, i is active number of words in memory (counting from position 0) and s is the content of the stack. In contrast to traditional programs, the effect of a smart contract execution can only be a manipulation of the blockchain's world state, but cannot depend on any input or output external to the blockchain. This way, determinism of execution is guaranteed, in contrast to other (permissioned) blockchains like Hyperledger Fabric where

		Out-of-bound behaviour	
Integer operation	**In-bounds requirement**	**EVM [38]**	**Solidity [32]**
$x +_s y, x -_s y, x \times_s y$	x_∞ op $y_\infty \in [-2^{n-1}, 2^{n-1} - 1]$	modulo 2^{256}	modulo 2^n
$x +_u y, x -_u y, x \times_u y$	x_∞ op $y_\infty \in [0, 2^n - 1]$	modulo 2^{256}	modulo 2^n
$x /_s y$	$y \neq 0 \wedge (x \neq -2^{n-1} \vee y \neq -1)$	0 if $y = 0$ -2^{255} if $x = -2^{255} \wedge y = -1$	0^* / INVALID† if $y = 0$ -2^{n-1} if $x = -2^{n-1} \wedge y = -1$
$x /_u y$	$y \neq 0$	0	0^* / INVALID†
$x\ mod_{s/u}\ y$	$y \neq 0$	0	0^* / INVALID†

$*$ Solidity version $<$ 0.4.0; $\dagger$ Solidity version $\geq$ 0.4.0

Table 1: Behaviour of integer operations in EVM and Solidity. Both x and y are n-bit integers, where x_∞, x_∞ denote their ∞-bit mathematical integers.

non-determinism is possible and may prevent the system from reaching consensus.

2.1.2 Memory model. EVM uses a memory model that is specific to the execution of smart contracts on a blockchain and differs from the traditional von Neumann architecture, which may cause confusion for novice developers. Instead of organising code, heap and stack in one large general-purpose memory, EVM features four different types of memory, whereas each has different properties and usage costs in terms of gas. EVM bytecode is stored by transactions and cannot be changed after it has been committed to the blockchain. Instructions operate on a *stack* of 256-bit big-endian words. The stack is private to a single contract (but not to methods within the contract) and is almost free to use in terms of gas. The size of the stack is limited to 1024 items. In addition to the stack, smart contracts can store variables in a *memory*. The memory is a random-access array of 256-bit words that is accessible only by the currently executed smart contract. Memory is always initialised with zeros and thus isolated from previous executions. Besides stack and memory, EVM features a *storage*. While stack and memory are volatile and only hold values during execution of a contract, storage is persistent and part of the world state. It is organised as a Patricia Merkle trie holding sets of persistent key/value pairs of all accounts. Storage is isolated from the other smart contracts and is the only way for a smart contract to save values across executions.

2.2 The Solidity Programming Language

Solidity is currently the most prominent programming language for developing smart contracts in Ethereum. Its syntax resembles a mixture of C and JavaScript, but it comes with a variety of unique concepts that are specific to smart contract development and might be unfamiliar for new developers, such as visibility modifiers[1] or the function-wide scoping of variables. In the context of this paper, the integer type handling of Solidity is most relevant. Solidity suggests to be a statically typed language, i.e. the compiler expects type information for each variable to be made explicit. Integers can be signed and unsigned, and of lengths between 8 and 256 bits in 8-bit steps denoted as `uint8` or `int128`, for instance. This resembles integer types in C and may lead novice developers to assume that a `uint8` will occupy 8 bits in memory, while an int128 occupies 128 bits. However, this assumption is false. Any integer type will be represented in the EVM by 256 bit big endian using two's-complement. That is, the integer type system of Solidity is not entirely consistent with that of EVM, which can lead to programming errors, as we show in our work. Explicit conversion between primitive types is possible, but the effects are not well documented (in fact, the documentation[2] reads: *Note that this may give you some unexpected behaviour so be sure to test to ensure that the result is what you want*). For example, explicitly casting a signed negative integer into an unsigned one will not result in the absolute value, but rather simply leave its bit-level representation intact.

2.3 Integer Bugs in Ethereum Smart Contracts

There are a multitude of different scenarios of integer operations that may result in bugs in Ethereum smart contracts [14]. We describe three main classes of integer bugs that may allow a malicious user to steal ether or modify the execution path of a smart contract: 1) *arithmetic bugs*, 2) *truncation bugs* and 3) *signedness bugs*. All three classes of bugs occur due to the mismatch between machine arithmetic and arithmetic over unbounded integers.

Arithmetic Bugs. We consider bugs such as integer overflows and underflows, but also bugs due to division by zero or modulo zero, as arithmetic bugs. Integer overflows (or underflows) occur when an arithmetic expression results in a value that is larger (or smaller) than it can be represented by the resulting type. The usual behaviour in such a case is to silently "wrap around", e.g. for a 32-bit type, reduce the value modulo 2^{32}. In C/C++ the out-of-bounds behaviour of integer operations is mostly undefined, whereas in Ethereum all behaviour is well-defined. Table 1 summarises the different out-of-bound behaviours enforced by the EVM and by Solidity. There are two noteworthy observations. Firstly, even though all arithmetic operations performed by the EVM are modulo 2^{256}, in Listing 1 the result of $a + b$ will silently wrap around if the value is larger than $2^{32} - 1$. This behaviour is enforced by Solidity, not by the EVM. Secondly, division (or modulo) by zero results in 0. In other programming languages this would result in an exception. However, in the EVM and Solidity versions prior to 0.4.0 this results in 0. Since most developers would expect an exception, starting from version

[1]external, public, internal, private

[2]http://Solidity.readthedocs.io/en/develop/types.html#conversions-between-elementary-types

0.4.0 the Solidity compiler injects an invalid operation to throw an assert-style exception, causing the EVM to revert all changes.

```
function add(uint32 a, uint32 b) public returns(uint) {
    return a + b;
}
```

Listing 1: An example of an overflow bug.

Truncation Bugs. Converting a value of one integral type to a narrower integral type which has a shorter range of values may introduce so-called *truncation bugs*. Truncation bugs became infamous due to a 64-bit integer value that was converted into a 16-bit integer, which ultimately led to the explosion of an Ariane 5 rocket in 1996. While truncation bugs in smart contracts will (hopefully) not lead to explosions, they may lead nevertheless to a loss of precision, which ultimately may affect the loss of ether. For instance, consider the default function in Listing 2. The function converts and stores the received amount of ether to an unsigned integer of 32 bits. `msg.value` is of type `uint`, which is equivalent to the type `uint256`, thus it can hold integer values ranging from 0 to $2^{256} - 1$. If a caller transfers an amount larger than $2^{32} - 1$, this value will be truncated and his balance will be lower than the amount that he effectively transmitted.

```
mapping(address => uint32) balance;

function() public payable {
  balance[msg.sender] = uint32(msg.value);
}
```

Listing 2: An example of a truncation bug.

Signedness Bugs. Lastly, converting a signed integer type to an unsigned type of the same width (or vice versa) may introduce so-called *signedness bugs*. This conversion may change a negative value to a large positive value (or vice versa). For example, consider the `withdrawOnce` function in Listing 3. This function allows a caller to withdraw only once a maximum amount of 1 ether from the smart contract's current balance. However, if the parameter `amount` is a negative value, it will pass the bounds check, be converted to a large unsigned integer and finally be passed as parameter to the `transfer` function. As a result, the `transfer` function will transfer an amount larger than 1 ether to the caller.

```
function withdrawOnce(int amount) public {
  if (amount > 1 ether || transferred[msg.sender]) {
    revert();
  }
  msg.sender.transfer(uint(amount));
  transferred[msg.sender] = true;
}
```

Listing 3: An example of a signedness bug.

3 METHODOLOGY

As we aim to detect integer bugs at the bytecode level, there are a number of challenges to overcome. In this section, we describe our approach towards inferring integer types, detecting integer bugs, applying taint analysis to reduce false positives and other challenges such as identifying intended checks for integer bugs.

3.1 Type Inference

Type information about integers such as size (e.g. 32 bits for `uint32`) and signedness (e.g. *signed* for `int`) are essential in order to check whether the result of an integer operation is in-bound or out-of-bound. However, type information is usually only available at the source code level and not at the bytecode level. That being said, due to certain code optimisations introduced by the Solidity compiler during compile time, it is actually possible to infer the size and the sign of integers at the bytecode level. For unsigned integers, the compiler introduces an AND bitmask in order to "mask off" bits that are not in-bounds with the integer's size. A zero masks the bit, whereas a one leaves the bit as it is. For instance, a `uint32` will result in an AND using `0xffffffff` as its bitmask. Thus, from the AND we infer that it is an unsigned integer and from the bitmask we infer that its size is 32 bits. For signed integers, the compiler introduces a sign extension via the SIGNEXTEND opcode. A sign extension is the operation of increasing the number of bits of a binary number while preserving the number's sign and value. In two's complement, this is achieved by appending ones to the most significant side of the number. The number of ones is computed using $256 - 8(x + 1)$, where x is the first value passed to SIGNEXTEND. For instance, an `int32` will result in a SIGNEXTEND using the value 3 as its first parameter. Thus, from the SIGNEXTEND we infer that it is a signed integer and from the value 3 we infer that its size is 32 bits, by solving the following equation: $8(3 + 1)$.

3.2 Finding Integer Bugs

We now describe the techniques we use for finding the three types of integer bugs described in Section 2.3.

Arithmetic Bugs. For each arithmetic instruction that could potentially overflow (i.e. ADD and MUL) or underflow (i.e. SUB), we emit a constraint that is only satisfied if the in-bounds requirements are not fulfilled (see Table 1). As an example, if we have an addition of two unsigned integers a and b, we emit a constraint to the solver that checks if $a + b > 2^n - 1$, where n denotes the largest size of the two values, e.g. in case a is a `uint32` and b is a `uint64`, n will be 64. Similarly, for signed/unsigned division (i.e. SDIV and DIV) and signed/unsigned modulo (i.e. SMOD, MOD, ADDMOD and MULMOD), we check whether the in-bounds requirements are not fulfilled as defined in Table 1. As an example, for signed division we emit a constraint that checks whether the divisor can be zero. If the solver can satisfy any of the emitted constraints under the current path conditions, we know that an arithmetic bug such as an overflow or a division by zero is possible.

Truncation Bugs. Solidity truncates signed and unsigned integers using AND and SIGNEXTEND instructions, respectively. For each instruction, we check whether it is possible for the input to be outside the range of the output. We do this by adding a constraint to the solver that is satisfied when the input value is larger than the output value. Moreover, we check the truncator value against two patterns, in order to detect and ignore truncations that have been intentionally introduced by Solidity. First, we check whether the binary representation of the truncator is equivalent to 160 ones. This represents a conversion to the type `address`. The second pattern

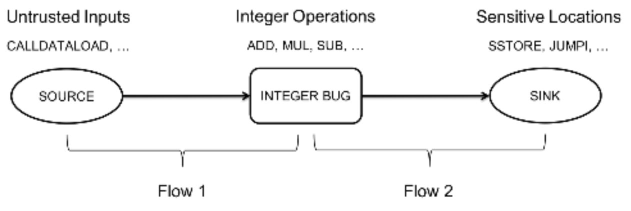


Figure 1: An integer bug is reported as valid iff it originates from a source and flows to a sink.

consists in checking whether the binary representation of the truncator contains any zeros (ignoring leading zeros). This pattern aims at filtering out truncations that have been introduced by Solidity in order to squeeze multiple variables in one data storage slot.

Signedness Bugs. We reuse the approach by Molnar et al. [21], and adapt it to detect signedness bugs in Ethereum smart contracts. The idea is to reconstruct signed/unsigned type information on all integral values, from the executed EVM instructions. This information is present in the source code but not in the bytecode. The algorithm to infer this information automatically works as follows: consider four different types for integer values: "Top", "Signed", "Unsigned", "Bottom". "Top" means the value has not been observed in the context of a signed or unsigned integer; "Signed" means that the value has been used as a signed integer; "Unsigned" means the value has been used as an unsigned integer; and "Bottom" means that the value has been used inconsistently as both, a signed and unsigned integer. These types form a lattice of four points. Our goal is to find symbolic variables that have type "Bottom". Every variable starts with the type "Top". During execution we modify the type of a variable, based on the type constraints of certain instructions. For example, a signed comparison (e.g. SLT, SGT, etc.) between two variables causes both variables to receive the type "Signed", whereas an unsigned comparison (e.g. LT, GT, etc.) between two variables causes both variables to receive the type "Unsigned". Any variable that received both a signed and unsigned type, receives the type "Bottom".

3.3 Taint Analysis

Taint analysis is a technique that consists in tracking the propagation of data across the control flow of a program. Taint analysis is extensively being used by numerous integer error detection tools in order to reduce the number of false positives [3, 29, 36, 37]. It is certainly possible to detect integer bugs without taint analysis. However, there are cases where integer bugs might be benign. For example, the Solidity compiler injects during compilation time integer overflows at certain locations in the bytecode in order to optimise it for later execution. These overflows are intentional and should not be flagged as malicious. Taint analysis can help to distinguish between benign overflows introduced by the developer or compiler, and malicious overflows that are exploitable by an attacker. In taint analysis we have the notion of so-called *sources* and *sinks*, with the idea that data originates from a source and eventually flows into a sink. Taint is introduced by sources, which is subsequently propagated across the state of a program. In the case of the EVM, the program state consists of the stack, memory and storage. We follow a very precise approach on how taint should be propagated across stack, memory and storage, by taking the exact semantics of every EVM instruction into account (see Section 4.2). Sources are locations in a program, where data is originating from an untrusted input that might be controllable by an attacker, for example, environmental information or function parameters. Sinks represent locations, where data is used in a sensitive context, for example, security checks or access to storage. Thus, the attack surface of a smart contract is defined by the EVM instructions that are exposed to an attacker. In other words, an attacker is limited to certain sources in order to trigger bugs that are used in sensitive sinks. Therefore, by deliberately ignoring integer bugs that do not originate from a source and do not flow into a sink, we can focus exclusively on actual exploitable integer bugs and gracefully reduce the number of false positives, Figure 1 illustrates this process. We only check for integer bugs where the input data to the integer operation is tainted. Finally, we only validate an integer bug if it flows into a sink.

Sources. There are a number of EVM instructions, which an attacker could potentially use in order to introduce data that might lead to the exploitation of integer bugs. These instructions can be divided in: 1) block information, such as GASLIMIT or TIMESTAMP), 2) environmental information, such as CALLER or CALLDATALOAD and 3) stack, memory, storage and flow operations, such as SLOAD or MLOAD. However, many of these instructions have certain requirements and limitations which makes them almost impossible to be used by attackers in practice. For example, block information such as the TIMESTAMP can only be introduced by a miner and the proposed value may only have a divergence of 15 seconds from the timestamp of the other miners. Another example of a limited instruction, is environmental information, such as the CALLER. An attacker can generate as many accounts as he wants, but he can not predict the value of the account address. Thus generating a desired address is essentially the same as brute-forcing. Therefore, we selected CALLDATALOAD and CALLDATACOPY as sources for our taint analysis. The reasons are twofold, first, an attacker can pick any arbitrary value (he is solely limited by the data type chosen by the developer) and second, the values are introduced at the transaction level and are therefore not only limited to miners.

Sinks. Whether or not an integer bug is harmful depends on where and how the smart contract uses the affected integer value. Such sensitive locations may originate from 1) stack, memory, storage and flow operations such as SSTORE or JUMPI and 2) system operations such as CREATE or CALL. We selected SSTORE, JUMPI, CALL and RETURN as sinks for our taint analysis, as these opcodes have an impact on path execution, storage and the sending of ether.

3.4 Identifying Benign Integer Bugs

Although, taint analysis already reduces significantly the number of false positives, there are still some cases where an integer bug might originate from an untrusted source and flow into a sensitive sink, while being a benign integer bug. In order to cope with such cases, we came up with some heuristic rules that allow us to detect specific cases of benign integer bugs. For example, Instead of immediately

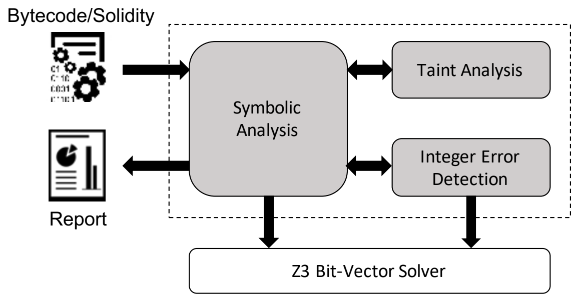


Figure 2: Architecture overview of Osiris. The shaded boxes represent its main components.

reporting an integer overflow or underflow as valid when we find it to be part of a branch condition, we check whether the predicate is designed to actually catch the bug. We note that common checks make use of the erroneous result to catch integer overflows and underflows, for example `if ((x + 1) < x)` or `if (x != (x * y) / y)`. We observe that these checks often use the same variable, on the right-hand side as well as on the left-hand side of the predicate. We also observe that if a predicate catches an integer bug, it is inclined to return soon or jump to a uniform error handling function. Hence, we report an integer bug as invalid, if we find the predicate to use the same variable, on the right-hand side as well as on the left-hand side, and one successor block of the branch condition in the control flow graph ends in a `JUMPI`, `REVERT` or `ASSERTFAIL`.

4 OSIRIS

In this section, we provide an overview on the overall design and implementation details of Osiris[3].

4.1 Design Overview

Figure 2 depicts the architecture overview of Osiris. Osiris can take as input the bytecode or Solidity source code of a smart contract. The latter gets internally compiled to EVM bytecode. Osiris outputs whether a contract contains any integer bug (e.g. overflow, underflow, truncation, etc.). Osiris consists of three main components: *symbolic analysis*, *taint analysis* and *integer error detection*. The symbolic analysis component constructs a Control Flow Graph (CFG) and symbolically executes the different paths of the contract. The symbolic analysis component passes the result of every executed instruction to the taint analysis component as well as to the integer error detection component. The taint analysis component introduces, propagates and checks for taint across stack, memory and storage. The integer error detection component checks whether an integer bug is possible within the executed instruction.

4.2 Implementation

We implemented Osiris on top of Oyente's [20] symbolic execution engine. Oyente faithfully simulates 124 out of the 134 EVM bytecode instructions. The non-faithfully simulated instructions consist of logging operations (i.e. `LOG0`, `LOG1`, `LOG2`, `LOG3` and `LOG4`), operations regarding the output data from a previous contract call (i.e. `RETURNDATASIZE` and `RETURNDATACOPY`), the operation to create a new contract (i.e. `CREATE`) and operations to call other contracts (i.e. `DELEGATECALL` and `STATICCALL`). Non-faithfully simulated means that the engine faithfully simulates the stack, but does not implement the complete logic of the operation as described in [38]. However, since all of these operations (except the logging operations) are related to contract calls and detecting integer bugs across contract calls is out-of-scope for this paper, we can safely ignore the non-faithfully simulated instructions by Oyente. Osiris is written in Python with roughly 1,200 lines of code (not counting Oyente's symbolic execution engine). In the following, we briefly describe the implementation of each main component.

Symbolic Analysis. The symbolic analysis component starts by constructing a CFG from the bytecode, where nodes in the graph represent so-called basic blocks and edges represent jumps between individual basic blocks. A basic block is a sequence of instructions with no jumps going in or out of the middle of the block. Osiris can output a visual representation of the CFG depicting the individual path conditions and highlighting the basic blocks that include integer bugs (see Figure 6 in Appendix A). After constructing the CFG, the symbolic execution engine starts by executing the entry node of the CFG. The engine consists of an interpreter loop that gets a basic block as input and symbolically executes every single instruction within that basic block. The loop continues until all the basic blocks of the CFG have been executed or a timeout is reached. In the case of a branch, the symbolic execution engine queries Z3 [4] in order to determine which path is feasible. If both paths are feasible, then the symbolic execution engine explores both paths in a Depth First Search (DFS) manner. Loops are terminated once they exceed a globally defined loop limit.

Taint Analysis. The taint analysis component is responsible for *introducing*, *propagating* and *checking* of taint. The symbolic execution engine forwards every executed instruction to the taint analysis component. Afterwards, the taint analysis component checks wether the executed instruction is part of the list of defined sources. If that is the case, the taint analysis component introduces taint by tagging the affected stack, memory or storage location. We faithfully introduce and propagate taint across stack, memory and storage. We implemented the stack using an array structure following LIFO logic. To represent memory and storage, we simply used a Python dictionary that maps memory and storage addresses to values. Since the EVM is a stack-based and register-less virtual machine, the operands of an instruction are always passed via the stack. Our taint propagation method identifies the operands of each EVM bytecode instruction and propagates the taint according to the semantics of each instruction as defined in [38]. The taint propagation logic tags according to the following principle: if an instruction uses a tainted value to derive another value, then the derived value becomes tainted as well. By following this principle, we achieve a more precise taint propagation than, for instance, Mythril [23]. Mythril propagates taint across the stack, but for certain instructions it does not propagate taint across memory or storage. For example, the instruction `SHA3` computes the Keccak-256 hash over a memory region that is determined by two operands that are pushed onto the stack: *offset* and *size*. Mythril simply checks if at least one of the two operands is tainted. If so, it taints

[3]Osiris is available at https://github.com/christoftorres/Osiris

the result that is pushed onto the stack. Osiris on the other hand, does not check the operands, but the memory region. Osiris only taints the result, if at least one of the values, that is stored in the given memory region, is tainted. As a final step, the taint analysis component verifies if a taint flow occurred, by checking whether the executed instruction is part of the list of defined sinks and if any of the values it used has been tainted by an integer bug.

Integer Error Detection. In contrast to the taint analysis component, the integer error detection component is not called upon every executed instruction. The integer error detection component is only called at instructions that may result in integer bugs, such as arithmetic instructions. For example, integer overflow checks are only performed if the symbolic analysis component executes an `ADD` or a `MUL` instruction, whereas width conversion checks are only performed if the symbolic analysis component executes an `AND` or a `SIGNEXTEND` instruction. Moreover, calls to the integer error detection component are only performed if at least one of the operands of the executed instruction is tainted. Iff these criteria are met, then the symbolic execution engine eventually forwards the executed instruction along with the current path conditions to the integer error detection component. Afterwards, the component follows the different techniques as described in Section 3.2 in order to detect the specific integer bugs. For example, in the case of an `AND` instruction with tainted operands, the symbolic analysis component will call the integer overflow detection method of the integer error detection component. The integer overflow detection method first tries to infer the sign and the width of the two integer operands as described in Section 3.1 and then creates a formula with a constraint that is only feasible if an integer overflow is possible under the current path conditions. This formula is afterwards passed on to the Z3 solver, which checks for its feasibility. If the solver finds a solution to the formula, then the integer error detection component knows that an integer overflow is possible and returns an error back to the symbolic analysis component. After that, the symbolic analysis component calls the taint analysis component, which then taints the result of the `AND` instruction where its source represents the discovered integer bug.

5 EVALUATION

In this section we assess the correctness and effectiveness of Osiris via an empirical analysis and demonstrate its usefulness in detecting real-world vulnerabilities in Ethereum smart contracts. The empirical analysis is separated in a qualitative and a quantitative analysis. Via the qualitative analysis we aim to determine the reliability of our tool by comparing our results with Zeus [18]. Via the quantitative analysis we intend to demonstrate the scalability of Osiris and to measure the overall prevalence of integer bugs contained in smart contracts that are currently deployed on the Ethereum blockchain.

Experimental Setup. All experiments were conducted on our high-performance computing cluster using 10 nodes with 960 GB of memory. Every node has 2 Intel Xeon L5640 CPUs with 12 cores each and clocked at 2,26 GHz, running a 64-bit Debian GNU/Linux 8.10 (jessie) with kernel version 3.16.0-4. We used version 4.6.0 of Z3, as our constraint solver for the symbolic execution engine as well as for our integer error detection module. For the symbolic execution engine we set a timeout of 100 ms per Z3 request. The global timeout for the symbolic execution was set to 30 minutes per contract. For our integer error detection module we set a timeout of 15 seconds per Z3 request. The loop limit, depth limit (for DFS) and gas limit for the symbolic execution engine was set to 10, 50 and 4 million, respectively.

Tool	Safe	Unsafe	No Result	Timeouts
Osiris	711	172	0	35
Zeus [18]	233	628	22	14

Table 2: Number of integer overflows and underflows detected by Zeus and Osiris.

5.1 Empirical Analysis

5.1.1 *Qualitative Analysis.*

Dataset. Kalra et al. [18] present a tool called Zeus, which is capable of detecting integer overflows and underflows. The authors evaluate their tool using a dataset of 1,524 contracts that they obtained by periodically scraping explorers such as Etherscan, Etherchain and EtherCamp over a period of three months [19] on the main and test network. We decided to reuse this dataset in order to compare our results with Zeus and to evaluate bugs that Zeus does not detect such as division by zero or truncation bugs. However, the published dataset does not contain any bytecode or source code. Eventually, we were able to download the bytecode and source code for 961 contracts, where 883 are unique.

Results. We run Osiris on the 883 contracts and summarise our results for each of the three types of bugs below.

Arithmetic Bugs. We compare Osiris's capability of detecting integer overflows and underflows with Zeus. Table 2 shows that Osiris reports most contracts to be safe whereas Zeus reports most contracts to be unsafe. "Safe" means that no overflow or underflow has been detected, whereas "unsafe" means that either an overflow or an underflow has been detected. The reason for discrepancy between Zeus and Osiris, is that Osiris aims at detecting solely overflows and underflows that are exploitable by an attacker in practice, thus limiting the number of reported bugs, while Zeus aims to be complete. Zeus reports no result for 22 contracts, where no result means either an error occurred or a timeout. Zeus encountered less timeouts than Osiris, with 14 compared to 35. On the other hand, Osiris managed to always faithfully return a result.

Table 3 depicts the confusion matrix of the evaluation between Osiris and Zeus. Osiris reports 5 contracts to be unsafe, whereas Zeus reports them to be safe. We manually verified these 5 contracts and indeed found them to potentially produce integer overflows. Listing 4 provides an example of a vulnerable function contained in one of the 5 contracts. The multiplication in the function `convertToWei` may overflow if `amount` is large enough. This questions Zeus claim to be sound in terms of achieving zero false negatives. In 471 cases, Zeus reports a contract to be unsafe while Osiris reports it to be safe. We manually analysed these cases and found that in some cases overflows were benign. These benign overflows

		Osiris		
		Safe	Unsafe	No Result
Zeus [18]	Safe	228	5	0
	Unsafe	471	157	0
	No Result	12	10	0

Table 3: Comparison between Zeus and Osiris.

were induced by the developer or by the Solidity compiler as part of handling data structures of dynamic size such as arrays, strings or bytes. The remaining overflows were indeed possible overflows, that were not caught by Osiris. Osiris could not catch them because they do not originate from the sources that we defined. So technically Osiris could detect them by adding more sources, such as loading from storage (i.e. SLOAD). Apart from that, the authors of Zeus state in their paper that for several cases their tool reported unsafe, although the contract was safe. We encountered 32 of these cases. Osiris reports 28 of these cases to be safe, thus about 88% less than Zeus. Unfortunately, Zeus does not check for division by zero or modulo zero bugs, thus we cannot compare Osiris to Zeus in this regard. Osiris did not find any modulo bugs. However, it did find 26 contracts vulnerable to division by zero bugs. We confirm the results via manual analysis of the source code and verifying that the bytecode was compiled using a compiler version lower than 0.4.0.

```
convertToWei(uint amount, string unit) external
     constant returns (uint) {
  return amount * etherUnits[unit];
}
```

Listing 4: Overflow in *EtherUnitConverter*'s `convertToWei` function, not detected by Zeus.

Truncation Bugs. Osiris reports 39 contracts carrying truncation bugs. We manually verified the findings and confirm the 39 bugs to be true positives. To confirm the findings, we checked the source code for type castings where integers are converted to smaller ranges.

Signedness Bugs. Signedness bugs seem to be less common. Osiris only reports 6 contracts to be vulnerable. Also here, we manually verified the findings and confirm the 6 bugs to be true positives. In order to confirm, we looked for conversions between signed and unsigned integers in the source code.

5.1.2 Quantitative Analysis.

Dataset. We collected the bytecode of 1,207,335 smart contracts, by downloading the first 5,000,000 blocks from the public Ethereum blockchain. The timestamps of the collected smart contracts range from August 7, 2015 04:42:15 AM to January 30, 2018 1:41:33 PM. Figure 3 depicts the number of smart contracts in our dataset with respect to the month of their deployment on the blockchain. We state a sudden increase of smart contracts, starting from April 2017. Ethereum does not store the source code of smart contracts. To obtain the source code of a smart contract, users often refer to services such as Etherscan. However, at the time of writing, Etherscan solely lists the source code of 29,486 smart contracts [6].

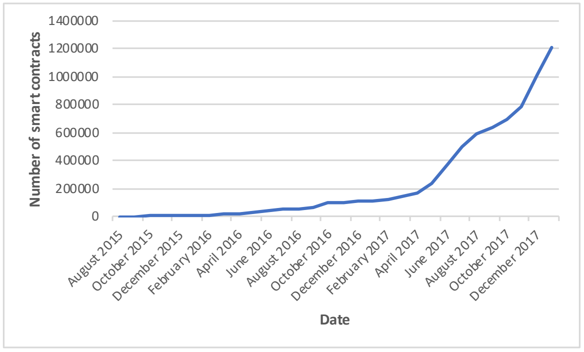


Figure 3: Number of smart contracts in Ethereum has increased abruptly.

Hence, only around 2% of the smart contracts on the Ethereum blockchain have their source code publicly available. Again, this emphasises the need for tools such as Osiris, that are capable of analysing smart contracts directly at the bytecode level. Out of these 1,207,335 contracts, only 50,535 are unique in terms of their bytecode. In other words, 96% of the smart contract on the Ethereum blockchain are just copies.

Performance. On average, Osiris takes 75 seconds to analyse a contract, with a median of 13 seconds and a mode of 1 second. 524 contracts require more than 30 minutes to analyse. The number of paths explored by Osiris ranges from 1 to 1394 with an average of 71 per contract and a median of 51. Similar to [20], we observe that the running time depends almost linearly on the number of explored paths. Finally, during our experiments, Osiris achieved a code coverage of about 88% on average.

Results. Figure 4 and Figure 5 report our results. Osris detects 42,108 contracts which contain at least one of the integer bugs discussed in Section 2.3. Out of these, 14,697 are distinct (by direct comparison of their bytecode). Figure 4 shows that most reported bugs are arithmetic (e.g. overflows, underflows, etc.) with 41,379 contracts as compared to 2,738 and 405 contracts for truncation and signedness, respectively. Out of these 41,379 contracts, 14,107 are found to be distinct, which account for roughly 28% of the 50,535 distinct contracts in our dataset. Figure 5 depicts the distribution between reported arithmetic bugs. We note that overflows are the most common type of bugs with 23,473 vulnerable contracts, where 10,520 are distinct which account for about 21% of the distinct contracts in our dataset. Immediately after that, follow underflows with 11,479 vulnerable contracts, where 6,103 are distinct which account for about 12% of the distinct contracts in our dataset. It is interesting to note that even though we only detect 29 distinct contracts vulnerable to modulo zero, the number of overall vulnerable contracts is 10,335. This implies that certain contracts are copied excessively and that one bug in such a contract, can have a huge impact on the security of thousands of other contracts on the blockchain.

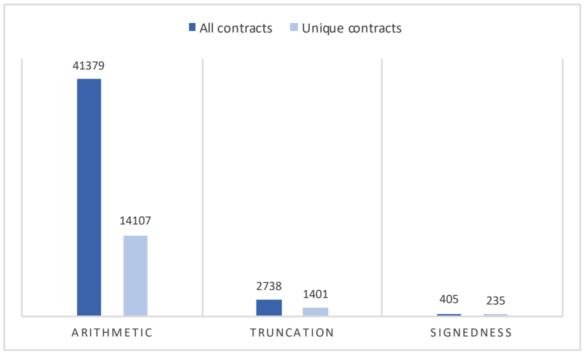


Figure 4: Number of vulnerable contracts reported by Osiris per integer bug.

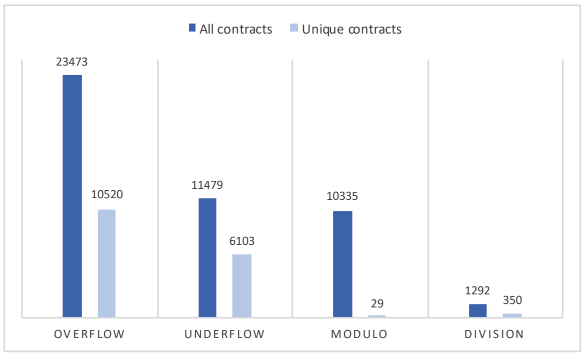


Figure 5: Number of vulnerable contracts per arithmetic error type.

5.2 Detection of Real-World Vulnerabilities

In this section, we examine the effectiveness and usefulness of Osiris in detecting and reporting real-world vulnerabilities. For this purpose, we run Osiris on five divulged vulnerabilities and analyse 495 top Ethereum token smart contracts.

5.2.1 *Detecting Known Vulnerabilities.*

Recently, a security company called Peckshield[4] disclosed five different vulnerabilities targeted at ERC-20 token smart contracts, each exploiting an integer overflow (see Table 4). Osiris successfully detects *all* the vulnerabilities listed in Table 4. From this small-scale experiment, we gain confidence that Osiris is suitable as a detection tool for vulnerabilities in real-world smart contracts.

[4]https://peckshield.com/

Token	Bug Name	CVE Number	Disclosed
BEC [5]	batchOverflow	CVE-2018-10299	22 April 2018
SMT [9]	proxyOverflow	CVE-2018-10376	25 April 2018
UET [11]	transferFlaw	CVE-2018-10468	28 April 2018
SCA [10]	multiOverflow	CVE-2018-10706	10 May 2018
HXG [8]	burnOverflow	CVE-2018-11239	18 May 2018

Table 4: CVEs examined by Osiris.

5.2.2 *Detecting Unknown Vulnerabilities.*

In the previous experiment we analysed Osiris's capability of effectively detecting known CVEs. In this experiment, we want to check whether Osiris is capable of detecting yet undiscovered vulnerabilities in Ethereum token smart contracts.

Dataset. Etherscan provides a list of top tokens ranked by their market capitalisation [7]. As of June 2018, the list holds a total of 509 different tokens. Out of these, 495 have their source code publicly available. We downloaded the bytecode as well as the source code for these 495 smart contracts and analysed them using Osiris.

Results. Osiris reported 164 contracts to be vulnerable, where 126 contracts were reported to contain overflows and 54 to contain underflows. We verified the findings via manual inspection of the source code. We found two overflows to be false positives and the rest of the findings to be indeed true positives. However, although all of the reported overflows/underflows being semantically possible, yet most of them are unlikely to be exploited in practice. The reason is twofold: 1) a large number of overflows and underflows may only be triggered by the owner of the smart contract and 2) a large number of overflows and underflows are due to implementations either not checking whether the balance of a receiver may overflow after a transfer (see Listing 5), or whether the value of the total supply may underflow before subtracting the amount of tokens to be burned.

```
1 function transfer(address _to, uint256 _value) returns
      (bool success) {
2   if (balances[msg.sender] >= _value && _value > 0) {
3     balances[msg.sender] -= _value;
4     balances[_to] += _value;
5     Transfer(msg.sender, _to, _value);
6     return true;
7   } else { return false; }
8 }
```

Listing 5: Overflow at Line 4 in *StandardToken*'s `transfer` function.

```
1 function burn(uint256 _value) returns (bool success) {
2   if (balances[msg.sender] < _value) return false;
3   balances[msg.sender] -= _value;
4   _totalSupply -= _value;
5   Burn(msg.sender, _value);
6   return true;
7 }
```

Listing 6: Underflow at Line 4 in function `burn`.

Nevertheless, two integer underflows reported by Osiris, have proven to be of particular interest. Let us consider the code snippet in Listing 7. The code originates from a token called *RemiCoin*[5]. Osiris reports that an integer underflow is possible at Line 11. The issue arises at the check at Line 7 (ironically commented as checking for allowance). The condition is not checking whether the amount is higher than the allowance, but whether the allowance is higher or equal to the amount. This is probably due to a simple copy-paste

[5]https://etherscan.io/token/0x7dc4f41294697a7903c4027f6ac528c5d14cd7eb

mistake, as some contracts have the exact same condition but return if the condition is false rather than when its true. Nevertheless, this subtle mistake has two tremendous consequences: 1) an attacker can transfer all the tokens from any address to another address of her own and 2) the attacker can provoke an underflow, hereby setting her allowance to any amount she desires. The same bug is also present in the UET token [11].

```
function transferFrom(address from, address to, uint
    value) returns (bool success) {
  //checking account is freeze or not
  if(frozenAccount[msg.sender]) return false;
  //checking the from should have enough coins
  if(balances[from] < value) return false;
  //checking for allowance
  if(allowed[from][msg.sender] >= value) return false;
  //checking for overflows
  if(balances[to] + value < balances[to]) return false;
  balances[from] -= value;
  allowed[from][msg.sender] -= value;
  balances[to] += value;
  // Notify anyone listening that this transfer took
      place
  Transfer(from, to, value);
  return true;
}
```

Listing 7: RemiCoin's `transferFrom` function allows an arbitrary user to steal tokens from another user.

RemiCoin (RMC) was released in 2017 and has a market capital of $27,520. Its founder/CEO is unknown. At its peak in October 2017, RemiCoin was traded for $1.82, whereas now its value has dropped to $0.0147. At the time of writing, 348 addresses hold RemiCoins and a total of 11,497 transfers have been made so far. We checked whether this bug has been exploited in the wild. We found multiple transactions resulting in integer underflows[6]. However, we miss evidence of these being targeted attacks as the victims are still left with a rather high amount of tokens. Since the bug results in transactions with a legitimate allowance being refused, we find it quite surprising that this bug has not been noticed so far. Demonstrating the above attack on the public blockchain is feasible. However, for ethical reasons we were reluctant to do so. Therefore, we demonstrate the attack on a copy of the smart contract that we deployed on the Ropsten test network[7] and created two test accounts: 1) `0xe9131d546bba6e233b0a19e504179dc61365a77f` and 2) `0x7e2a886f1ba5942cc7a3a53fc6fae94868e318a0`. We deployed the contract via the first account, hence making this account the holder of the total supply of tokens. Afterwards, we performed our attack by calling the `transferFrom` function and passing as arguments the address of the first account, the address of the second account and finally the total supply of tokens[8]. As a result, the second account now owns all of the tokens and its allowance was set from zero to a substantial amount.

6 DISCUSSION

In this section, we summarise weaknesses in the Ethereum ecosystem that lead to smart contracts that are prone to integer bugs. Further, we discuss possible remedies to prevent integer bugs from happening in smart contracts.

6.1 Causes for Integer Bugs

Weaknesses of Solidity and EVM. Solidity is a language that has been designed to lower the bar for developers entering the smart contract ecosystem. In that respect, its syntax resembles JavaScript, suggesting a dynamically typed scripting language, which in fact, it is not. Then again, during compilation from Solidity to EVM, the compiler warns about some type casts which gives the developer the impression of a strictly static type validation – which again is not true. In fact, Solidity compiles into statically EVM bytecode, but the type system of Solidity does not strictly map into that of EVM. For example, although integers with less than 256 do not exist in EVM, Solidity attempts to give the developer the impression of different integer types by providing respective type identifiers and generating wrap-around behaviour during compilation. This is a weakness because first, it suggests that developers could save memory by using shorter integer types and second, it makes the unexpected (integers wrapping around) the rule.

As a second weakness we consider the overflow handling of EVM itself. Unlike in low-level programming, deliberate integer overflows is a rarely used feature in application development and we have not come across a single smart contract that uses integer overflow in a deliberate way. Nevertheless, neither the Solidity compiler nor EVM treat integer overflows as an exception but rather treat them as a real CPU would do – with some unexpected deviations such as feasibility of division by zero. Given the fact that aborting a smart contract will result in a safe rollback of the transaction, treating overflows as an exception and panicking seems to be the safer alternative than silent wrapping.

Unsafe Implementations of Standards. The ERC-20 [35] token standard provides a standardised Application Programming Interface (API) for tokens within Ethereum smart contracts. The API provides basic functionality in order to transfer tokens, as well as to allow tokens to be approved such that they may be spent by another on-chain third-party. The standard describes an interface consisting of a number of functions and events, which a smart contract must implement in order to be compliant. The main issue with the current standard, is that it solely provides an interface. Its implementation is left to the developer of the token. As a consequence, many different implementations exist. Some implementations might have bugs and might be copied by other developers, with the bugs left unnoticed, hereby spreading the bugs across multiple contracts. In addition, some tokens introduce new functionality that is not part of the standard and hereby potentially introduce new bugs.

Negligible and Incorrect Use of Safe Libraries. In Section 5.2.2 we analysed the safety of 495 token smart contracts. Token contracts perform a number of arithmetic operations such as subtracting from balances and adding to balances. However, these operations may produce integer bugs such as overflows and underflows. Therefore, it is recommended to perform such operations using a safe arithmetic library such as *SafeMath* [27]. SafeMath provides safe arithmetic operations for multiplication, division, addition and subtraction. We found that 337 out of the 495 contracts include the

[6]https://bit.ly/2LHeNf6
[7]https://bit.ly/2HIKbrx
[8]https://bit.ly/2l7ITNy

SafeMath library in their source code. Thus roughly 32% of the tokens do not make use of a SafeMath library and are therefore highly susceptible to overflows and underflows. Moreover, Osiris found 53 out of the 337 contracts to include bugs related to overflows and underflows. After manual inspection, we found that even though developers make use of the SafeMath library, this does not necessarily mean that they use it for every single arithmetic operation performed by their smart contract.

6.2 Ways Towards Safe Integer Handling

There are various ways to reduce the likelihood of potentially catastrophic integer bugs in Ethereum smart contracts. We discuss two different ways in the following:

1) *Handle integer bugs at the application layer.* This is the approach taken by libraries such as SafeMath. This is already a best practice and the only way to avoid overflows without modifying the Solidity compiler or EVM. However, it comes at the price of additional EVM instructions which increases gas costs. Obviously not all developers see the benefit of using additional libraries for solving apparently simple arithmetic tasks.

2) *Handle integer bugs at the compiler level.* Compiler-generated overflow checks remove the burden from developers but still create additional overhead in terms of gas costs and runtime performance. Other languages such as Rust go a route that combines rigorous static checking with fail-fast at development time and defensive programming at runtime. This approach could be retrofitted to the Solidity compiler without affecting the language or the EVM themselves: as we have shown in this paper, static integer overflow checking of real-world smart contracts is feasible and could be integrated into the compiler to identify potential overflow bugs at development time (as it is done by [24] for C code, for example). By additional annotations such as `//@allow_overflow`, developers could explicitly mark variables that should be treated in an unsafe way to allow deliberate overflows. The drawback is obviously that still, generated EVM contains potential unnecessary and costly runtime checks.

7 RELATED WORK

In the past years, several approaches have been proposed in order to tackle the challenge of fully formalising reasoning about Ethereum smart contracts. Numerous attempts have been made in modelling the semantics of Ethereum smart contracts in state-of-the-art proof assistants [1, 2, 12, 15, 16, 30]. Bhargavan et al. propose to translate a subset of Solidity to F* for formal verification [2]. This is similar to the approach initially followed by the Solidity compiler of translating Solidity contracts into WhyML to generate formal proofs for the why3 framework [30]. A number of alternative translations of EVM bytecode to manual assisted proofs have been proposed, including proofs in Coq [16] and Isabelle/HOL [1, 15]. While these approaches enable formal machine-assisted proofs of various safety and security properties of smart contracts, none of them provide means for fully automated analysis.

As a result, a large number of automated tools have been proposed for ensuring correctness and safety of smart contracts [18, 20, 23, 25, 26, 33]. All of these tools are based on symbolically executing EVM bytecode. Luu et al. were the first to present a symbolic execution tool called Oyente [20]. The tool is capable of automatically detecting vulnerability patterns such as *transaction-ordering dependence*, *timestamp dependence*, *mishandled exceptions* and *reentrancy*. Nikolic et al. present Maian [25], a tool that builds up on Oyente and employs inter-procedural symbolic analysis as well as concrete validation in order to find and validate vulnerabilities on trace properties, such as *greedy*, *prodigal*, and *suicidal*, in Ethereum smart contracts. Tsankov et al. present Securify [33], a tool that first symbolically analyses a contract's dependency graph to extract semantic information and afterwards checks for violations of safety patterns. To enable extensibility, the tool permits new patterns to be specified via a designated domain-specific language. In any case, none of the aforementioned tools currently check for integer bugs in smart contracts.

Kalra et al. propose Zeus [18], a framework for automated verification of smart contracts using abstract interpretation and symbolic model checking, accepting user-provided policies. Zeus inserts policy predicates as `assert` statements in the source code, then translates everything to an intermediate LLVM representation, and finally invokes its verifier to determine assertion violations. The tool is capable of detecting integer overflows and underflows similar to Osiris, with the difference of Osiris working at the bytecode level and Zeus at the source code level. However, source code is not always available. Moreover, Zeus requires users to write policies to assert the security of smart contracts, which is sometimes not that trivial. Mueller et al. present Mythril [23], a security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis and control flow checking to detect a variety of security vulnerabilities. Mythril comes very close to the approach behind Osiris, with one of the differences being that Osiris uses a more precise and complete taint propagation logic while allowing users to define their own sources and sinks. Another difference is that Mythril treats every integer as a 256-bit integer and therefore does not detect an overflow if for example two 32-bit integers are being added, Osiris on the other hand tries to infer the width of every integer in order to precisely tell if an arithmetic operation can overflow or not. Finally, at the time of writing, Mythril seems to have issues in distinguishing between benign and malignant overflows and underflows [22]. Osiris effectively distinguishes between benign and malignant integer bugs. Ultimately, both Zeus and Mythril, fail to check for truncation bugs and signedness bugs, whereas Osiris does check for these two types of integer bugs.

8 CONCLUSION AND FUTURE WORK

Integer bugs are currently listed as one of the top 3 vulnerabilities in smart contracts [13]. We present the design and implementation of Osiris – a framework for detecting integer bugs in Ethereum smart contracts. Osiris leverages on symbolic execution and taint analysis. We compare Osiris with Zeus and show that Zeus is not sound. Osiris finds 5 contracts to be unsafe whereas as Zeus reports them to be safe. Moreover, in our evaluation Osiris reports less false positives than Zeus. Our evaluation on over 1.2 million Ethereum smart contracts indicates that about 4% of them might be vulnerable to at least one of the three integer bugs presented in this paper. Finally, using Osiris we discovered a yet unknown vulnerability in a couple of Ethereum tokens.

In future work, we plan to extend Osiris's taint analysis to also track taint across multiple contracts (inter-contract analysis) and across different method invocations (trace analysis). Moreover, we aim to switch to concolic execution using concrete values from the blockchain in order to validate and generate direct exploits. This may help us make Osiris's detection mechanism even more precise. Finally, we want to augment our evaluation on the security of Ethereum tokens. Etherscan lists over 90,000 ERC-20 based token smart contracts on the Ethereum blockchain. Hence, we only scratched the tip of the iceberg, by analysing only 495 of them.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [34] – see https://hpc.uni.lu.

REFERENCES

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. *CPP. ACM. To appear* (2018).

[2] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. 2016. Formal Verification of Smart Contracts. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*. ACM Press, New York, New York, USA, 91–96. https://doi.org/10.1145/2993600.2993611

[3] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. IntFinder: Automatically detecting integer bugs in x86 binary program. In *International Conference on Information and Communications Security*. Springer, 336–345.

[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[5] Etherscan.io. 2018. BeautyChainToken. Retrieved June 7, 2018 from https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

[6] Etherscan.io. 2018. Ethereum Contracts with Verified Source Codes. Retrieved June 8, 2018 from https://etherscan.io/contractsVerified

[7] Etherscan.io. 2018. Etherscan Token Tracker Page. Retrieved June 5, 2018 from https://etherscan.io/tokens

[8] Etherscan.io. 2018. HexagonToken. Retrieved June 7, 2018 from https://etherscan.io/address/0xb5335e24d0ab29c190ab8c2b459238da1153ceba#code

[9] Etherscan.io. 2018. SmartMeshICO. Retrieved June 7, 2018 from https://etherscan.io/address/0x55f93985431fc9304077687a35a1ba103dc1e081#code

[10] Etherscan.io. 2018. Social Chain. Retrieved June 7, 2018 from https://etherscan.io/address/0xb75a5e36cc668bc8fe468e8f272cd4a0fd0fd773#code

[11] Etherscan.io. 2018. UselessEthereumToken. Retrieved June 7, 2018 from https://etherscan.io/address/0x27f706edde3ad952ef647dd67e24e38cd0803dd6#code

[12] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[13] NCC Group. 2018. DASP - TOP 10. Retrieved June 15, 2018 from https://dasp.co/#item-3

[14] Yoichi Hirai. 2016. Exception on overflow - Issue #796 - ethereum/solidity. Retrieved June 10, 2018 from https://github.com/ethereum/solidity/issues/796#issuecomment-253578925

[15] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security*. Springer, 520–535.

[16] Yoichi Hirai. 2017. Ethereum Virtual Machine for Coq (v0.0.2). Retrieved June 12, 2018 from https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18

[17] PeckShield Inc. 2018. PeckShield Inc. - Advisories. Retrieved June 13, 2018 from https://peckshield.com/advisories.html

[18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. NDSS.

[19] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus Evaluation. Retrieved June 12, 2018 from https://docs.google.com/spreadsheets/d/12_g-pKsCtp3lUmT2AXngsqkBGSEoE6xNH51e-of_Za8/preview?usp=embed_googleplus#gid=1568997501

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, New York, New York, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[21] David Molnar, Xue Cong Li, and David Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs.. In *USENIX Security Symposium*, Vol. 9. 67–82.

[22] Bernhard Mueller. 2018. Detecting Integer Overflows in Ethereum Smart Contracts. Retrieved June 12, 2018 from https://bit.ly/2JIp9ea

[23] Bernhard Mueller. 2018. Smashing Ethereum Smart Contracts for Fun and Real Profit. (2018).

[24] Paul Muntean, Jens Grosklags, and Claudia Eckert. 2018. Practical Integer Overflow Prevention. *In IEEE TSE journal (under review)* (2018). https://arxiv.org/abs/1710.03720

[25] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. *arXiv preprint arXiv:1802.06038* (2018).

[26] Trail of Bits. 2018. Manticore - Symbolic execution tool. Retrieved June 12, 2018 from https://github.com/trailofbits/manticore

[27] OpenZeppelin. 2018. OpenZeppelin/openzeppelin-solidity. Retrieved June 12, 2018 from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

[28] Sergey Petrov. 2017. Another Parity Wallet hack explained. Retrieved June 13, 2018 from https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c

[29] Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, and Angelos D Keromytis. 2014. IntFlow: improving the accuracy of arithmetic error detection using information flow tracking. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 416–425.

[30] Christian Reitwiessner. 2018. Formal Verification for Solidity Contracts. Retrieved June 12, 2018 from https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts

[31] David Siegel. 2016. Understanding The DAO Attack. Retrieved June 13, 2018 from https://www.coindesk.com/understanding-dao-hack-journalists/

[32] Solidity. 2018. Solidity 0.4.24 documentation. Retrieved June 9, 2018 from http://solidity.readthedocs.io/en/v0.4.24/

[33] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *arXiv preprint arXiv:1806.01143* (2018).

[34] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. IEEE, Bologna, Italy, 959–967.

[35] Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20 Token Standard. Retrieved June 7, 2018 from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[36] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society. http://www.isoc.org/isoc/conferences/ndss/09/pdf/17.pdf

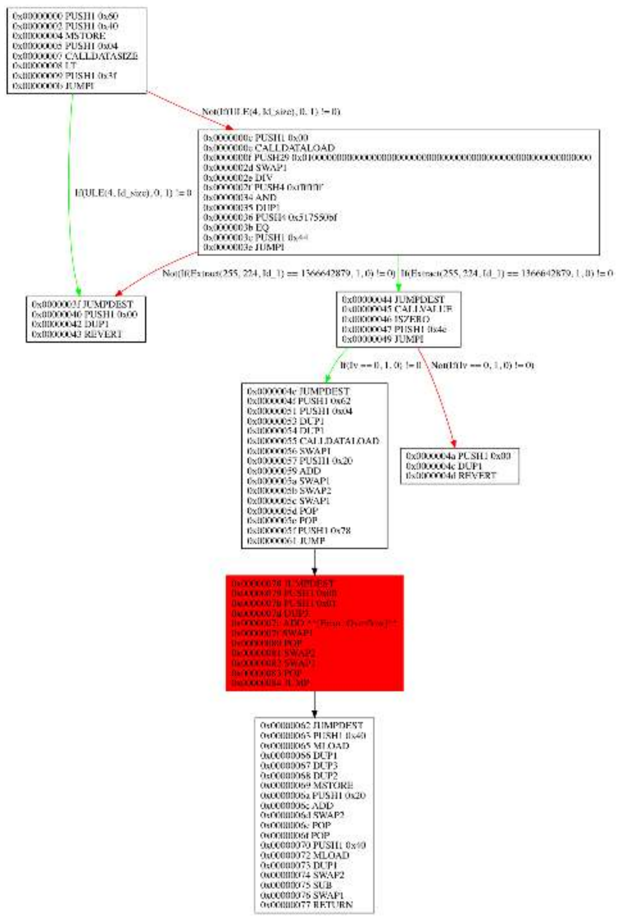
[37] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT.. In *OSDI*, Vol. 12. 163–177.

[38] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

A CONTROL FLOW GRAPH EXAMPLE

```
1 pragma solidity ^0.4.21;
2
3 contract Test {
4
5   function overflow(uint value) public pure returns(
        uint) {
6     return value + 1;
7   }
8 }
```

Listing 8: An example of a smart contract possibly producing an integer overflow at line 6.


Not(If(ULE(4, Id_size), 0, 1) != 0)
If(ULE(4, Id_size), 0, 1) != 0
Not(If(Extract(255, 224, Id_1) == 1366642879, 1, 0) != 0)
If(Extract(255, 224, Id_1) == 1366642879, 1, 0) != 0
If(Iv == 0, 1, 0) != 0
Not(If(Iv == 0, 1, 0) != 0)

```
0x00000000 PUSH1 0x60
0x00000002 PUSH1 0x40
0x00000004 MSTORE
0x00000005 PUSH1 0x04
0x00000007 CALLDATASIZE
0x00000008 LT
0x00000009 PUSH1 0x3f
0x0000000b JUMPI
```

```
0x0000000c PUSH1 0x00
0x0000000e CALLDATALOAD
0x0000000f PUSH29 0x0100000000000000000000000000000000000000000000000000000000
0x0000002d SWAP1
0x0000002e DIV
0x0000002f PUSH4 0xffffffff
0x00000034 AND
0x00000035 DUP1
0x00000036 PUSH4 0x517550bf
0x0000003b EQ
0x0000003c PUSH1 0x44
0x0000003e JUMPI
```

```
0x0000003f JUMPDEST
0x00000040 PUSH1 0x00
0x00000042 DUP1
0x00000043 REVERT
```

```
0x00000044 JUMPDEST
0x00000045 CALLVALUE
0x00000046 ISZERO
0x00000047 PUSH1 0x4e
0x00000049 JUMPI
```

```
0x0000004a PUSH1 0x00
0x0000004c DUP1
0x0000004d REVERT
```

```
0x0000004e JUMPDEST
0x0000004f PUSH1 0x62
0x00000051 PUSH1 0x04
0x00000053 DUP1
0x00000054 DUP1
0x00000055 CALLDATALOAD
0x00000056 SWAP1
0x00000057 PUSH1 0x20
0x00000059 ADD
0x0000005a SWAP1
0x0000005b SWAP2
0x0000005c SWAP1
0x0000005d POP
0x0000005e POP
0x0000005f PUSH1 0x78
0x00000061 JUMP
```

```
0x00000078 JUMPDEST
0x00000079 PUSH1 [illegible]
0x0000007b PUSH1 [illegible]
0x0000007d DUP3
0x0000007e ADD ***[Possible Overflow]***
0x0000007f SWAP1
0x00000080 POP
0x00000081 SWAP2
0x00000082 SWAP1
0x00000083 POP
0x00000084 JUMP
```

```
0x00000062 JUMPDEST
0x00000063 PUSH1 0x40
0x00000065 MLOAD
0x00000066 DUP1
0x00000067 DUP3
0x00000068 DUP2
0x00000069 MSTORE
0x0000006a PUSH1 0x20
0x0000006c ADD
0x0000006d SWAP2
0x0000006e POP
0x0000006f POP
0x00000070 PUSH1 0x40
0x00000072 MLOAD
0x00000073 DUP1
0x00000074 SWAP2
0x00000075 SUB
0x00000076 SWAP1
0x00000077 RETURN
```

Figure 6: A representation of the control flow graph that Osiris produces for Listing 8. The basic block highlighted in red indicates the location where an overflow may occur.

B THE DAO HACK

```
contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to){credit[to] += msg.value;}
  function queryCredit(address to) returns (uint){
    return credit[to];
  }
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

Listing 9: A simplified version of the DAO smart contract.

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;
  function Mallory2(){ owner = msg.sender; }
  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
  function() {
    if (performAttack) {
      performAttack = false;
      dao.withdraw(1);
  }}
  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

Listing 10: A more efficient attack than the original DAO attack.