2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)

A Survey of Tools for Analyzing Ethereum Smart Contracts

Monika di Angelo
Eurecom, Sophia Antipolis, France
Technische Universität Wien, Vienna, Austria
monika.diangelo@tuwien.ac.at

Gernot Salzer
Eurecom, Sophia Antipolis, France
Technische Universität Wien, Vienna, Austria
gernot.salzer@tuwien.ac.at

***Abstract*—Smart contracts are at the heart of many decentralized applications, encapsulating core parts of the business logic. They handle the exchange of valuable assets like crypto-currencies or tokens in a transparent, decentralized manner. Being computer programs, they are also prone to programming errors, which have already lead to spectacular losses. Therefore, methods and tools have emerged to support the development of secure smart contracts and to aid the analysis of deployed ones.**

Assessing the quality of such tools turns out to be difficult. There are academic tools, tools developed by companies, and community tools in open repositories, but no comprehensive survey that may serve as a guide. Most discussions of related work in research papers are not helpful either, as they concentrate on methods rather than tools, base their review on publications about the tools rather than the tools themselves, or disregard tools outside of academia.

Our survey aims at filling this gap by considering tools regardless of their provenance and by installing and testing them. It is meant as a guide for those who intend to analyze already deployed code, want to develop secure smart contracts, or plan to teach a related subject. We investigate 27 tools for analyzing Ethereum smart contracts regarding availability, maturity level, methods employed, and detection of security issues.

***Index Terms*—analysis, comparison, Ethereum, smart contracts, survey, tools**

I. Introduction

A decentralized application (dApp) typically consists of a front-end that interacts with the environment and a back-end that stores critical data via distributed ledger technology, e.g. on a blockchain. It may even outsource parts of the business logic to a so-called *smart contract*. Our survey concentrates on *Ethereum* as the most prominent platform for smart contracts.

It is imperative for smart contracts to function properly since bugs may lead, and indeed have lead, to tremendous losses and disruptions. Bugs even occur in contracts by experienced programmers, which underlines the fact that smart contract programming is tricky. Therefore, numerous methods and tools have emerged to support the development of secure smart contacts and to aid the analysis of already deployed ones.

Suppose you want to assess the state of the art in order to identify tools that you can actually employ, say, in a dApp project. A natural place to start is research papers with their discussions of related work. As it turns out, however, *these comparisons have several shortcomings*.

a) Academic papers concentrate on methods rather than tools. Tools of varying maturity, size, availability, and utility are presented on equal footing. Largely undocumented proof-of-concept implementations receive the same attention as well documented, highly functional tools.

b) Reviews often rely on publications by the authors of a tool, instead of basing them on the tool itself. As an anecdote, at least two conference papers, three master theses, some blogs and online paper collections report the tool *DappGuard*. Checking the source [1] and the repository on Github, it turns out to be the seminar assignment of three students who had the task of *envisioning a security tool without implementing it*, and indeed, it does not exist.

c) Academic surveys tend to disregard tools outside academia. As an example, one otherwise fine paper dismisses five tools by stating that "Other static analysis tools are available online (e.g. [...]), but they are not accompanied by any academic paper", even though information on the tools is available in other forms.

The present paper fills the gap by considering tools for smart contract analysis regardless of their provenance and by concentrating on the tools themselves. It is meant as a guide for those who intend to analyze already deployed code, want to develop secure smart contracts, or plan to teach a related subject. We investigate the availability of the tools as well as their functionality. Moreover, we compare their characteristics in a compact and structured manner.

Methodology. We compiled a comprehensive list of tools for analyzing smart contracts by checking the publications of the main conferences in the field and by following the references therein. Moreover, we searched the internet for additional tools and scanned Github for relevant projects. We installed all publicly available tools locally and checked their functionality by running them on some examples. We also looked at the source code for more information, such as code reuse and dependencies. We compared the tools with respect to several criteria, including methods, project size, and development dynamics.

978-1-7281-1264-0/19/$31.00 ©2019 IEEE
DOI 10.1109/DAPPCON.2019.00018

Limitations. At the time of writing (October 2018) the compilation of tools is comprehensive. However, as this is a lively field, updates and new tools will keep turning up. Moreover, we do not evaluate how well the tools achieve their goals.

Roadmap. Section II briefly explains the methods employed by the tools. Section III presents each tool individually with its defining characteristics. Section IV compares the tools with respect to various criteria. Section V summarizes our conclusions.

II. Methods Employed by the Tools

We briefly explain the fundamental methods and notions mentioned in this survey, within the context of Ethereum smart contracts.

Bytecode refers to the list of byte-size integers that serve as instructions for the Ethereum Virtual Machine (EVM).

Source code refers to a program in a high-level programming language, here Solidity.

Static analysis refers to a class of methods that examine the source code or bytecode of a contract without executing it. Most methods listed below are static.

Dynamic analysis means to observe a contract while executing (parts of) it in the original context.

Disassembling means to translate EVM bytecode into better readable assembly language, where machine operations and storage addresses are represented symbolically.

Decompilation is the process of transforming EVM bytecode to a more compact representation on a higher abstraction level (like intermediate or Solidity code) to enhance the readability of the code or to ease data flow analysis.

Basic block is a sequence of statements without branches.

Control flow graph (CFG) is a directed graph, where the basic blocks of a program serve as the nodes. An arc connects node A with node B if it is possible that block B gets executed immediately after block A. The arc may be labeled by the condition under which this path is chosen.

Dynamic CFG is similar to a CFG with the difference that arcs indicate the actual control flow encountered during a particular execution of the code.

Call graph is a directed graph, where the nodes are functions. There is an arc from node A to node B if function A calls function B.

Abstract Syntax Tree (AST) represents the syntactic structure of Solidity code as a tree. It occurs as an intermediate product when compiling Solidity to bytecode. Often, it is better suited for analyzing Solidity code.

Contextualization means the feedback about where in the Solidity or bytecode an issue occurred, either by indicating the line in the code or by identifying the affected function.

Execution trace is the sequence of instructions (possibly including additional information) executed during a particular run of the code.

Code instrumentation means to add instructions to the contract under analysis to monitor performance and to check assertions.

Transformation from a stack- to a register-oriented view is a particular decompilation technique that replaces stack-oriented instructions of the EVM by instructions operating on registers. Register-oriented view is not only easier to understand but also helps in the analysis of the data flow.

Constraint solving means to determine the solvability (a.k.a. satisfiability) of constraints and possibly to compute a concrete solution. A constraint is a set of conditions that variables have to satisfy. In our context, constraints mostly arise from branching conditions in the code (which are also used to label the arcs in CFGs). Depending on the operations in the constraints, constraint solving can be arbitrarily difficult. It is delegated to so-called SMT-solvers like Z3.

Symbolic execution means to execute code using symbols instead of concrete values for the variables. Operations on these symbols lead to algebraic terms, and conditional statements give rise to propositional formulas that characterize the branches. A particular part of the code is reachable if the conjunction of formulas on the path to this part is satisfiable, which can be checked by SMT-solvers.

Finite state machines (FSMs) are abstract models of systems that can be in a finite number of states only. FSMs are characterized by listing all states, by designating the initial and final states, and by describing the actions that will cause the machine to transition to another state.

Verification checks whether code meets the specification and fulfills its intended purpose.

Formal methods are mathematical techniques for specifying, developing, and verifying soft- and hardware.

Formal verification means verification by formal methods with the aim of proving or disproving system properties rigorously. As a prerequisite, all components referenced by such a property as well as their behavior must have been specified formally. E.g., to verify properties of smart contracts on bytecode level formally, we need a formal specification of the EVM and of the properties.

Model checking is a technique for automatically verifying correctness properties of finite-state systems. It requires a model of the system which is then checked against a given specification.

Specifying the EVM means to define the behavior of the EVM (its semantics) unambiguously. A *formal* specification additionally requires that the language used to specify the EVM is itself rigorously defined and does not admit any ambiguities. Then, properties of programs running on the EVM can be formally proven.

Horn logic is a restricted form of first-order logic where all formulas ('clauses') are if-then rules. Though restricted, Horn logic is still computationally universal, thus it can perform the same computations as any computer.

DataLog is a restricted form of Horn logic that is no longer computationally universal, but it allows for efficient processing, e.g. with tools like Soufflé.

Abstract interpretation ignores certain instructions or certain effects of instructions while executing the bytecode (abstracts them away). This can be done by translating instructions to another formalism (like DataLog) and then exploring all possible executions.

III. Tools for Analyzing Smart Contracts

In this section, we present tools for analyzing smart contracts in two groups according to their availability:

- Publicly available under an open source license
- Not publicly available, neither as source nor as binary

A. Tools Publicly Available

The first reference after the tool name points to the repository from where the tool can be downloaded.

ContractLarva [2], [3] performs runtime verification for smart contracts written in Solidity. For a given smart contract the user specifies its properties declaratively using dynamic event automata. With the specification and the contract as inputs, the tool generates a new Solidity contract that acts like the original one, but additionally contains code to check the runtime behaviour against the specification and to take compensatory actions in case of a violation. This command-line tool is available on Github under an Apache-2.0 license since December 2017. It requires Haskell and comes with a user manual.

E-EVM [4], [5] simulates the EVM visually. Starting from the disassembled bytecode, the tool executes the code symbolically, constructs a CFG, and displays the latter together with stack information. The back end of the tool consists of two almost identical Python scripts that are available on Github under an MIT license since January 2018. The front end (described in the paper) is not provided.

Erays [6], [7] is a tool for reverse engineering Solidity smart contracts. It disassembles the input bytecode, transforms it from a stack- to a register-oriented view, replaces chains of assignments by expressions, and annotates known function selectors (4 byte hashes) with the original function headers. The tool generates one pdf file per functional unit with its pseudo-code. It can be used to reverse-engineer contracts into a representation that is indeed easier to understand, which has also been demonstrated by the authors with several use cases. The Python scripts are operated from the command-line and are available on Github under an MIT license since August 2018. The tool depends on GraphViz for generating the pdf files. The documentation is minimal but sufficient.

EthIR [8], [9] transforms bytecode into an intermediate-level language suited as input to general purpose static analyzers. It is a modified version of the tool Oyente (see below) that disassembles the given bytecode and constructs a CFG. The basic blocks are transformed from a stack- to a register-oriented view. The control flow itself is represented as guarded rules. This rule-based representation can then be fed into SACO, a general purpose static analyzer by the same group, to deduce e.g. upper bounds for loops. This command-line tool is available on Github under a GPL-3.0 license since March 2018. It is written in Python and depends on particular versions of the SMT solver Z3, the Solidity compiler, and Go-Ethereum.

EtherTrust [10], [11] translates bytecode to Horn clauses that over-approximate its behaviour. Using the SMT solver Z3, the tool then checks properties like 'independence from transaction environment' and 'single entrancy'. This approach does not detect vulnerabilities, but provides guarantees that the code is free of certain ones. EtherTrust is a command-line tool written in Java and is available under a GPL-3.0 license since May 2018 as version 0.0.1. It is a proof-of-concept without documentation on how to interpret its output.

FSolidM [12], [13], [14] allows the user to specify the intended behaviour of a smart contract abstractly as a finite-state machine and then to generate automatically Solidity code implementing this behaviour. Plugins address problems like reentrancy or provide patterns for recurring functionality like access control. The correctness of the finite-state machine can be verified by specifying properties in temporal logic and proving them with the model checker nuXmv. The generated code is assumed to be correct by construction, but cannot be proven since Solidity lacks a formal semantics. FSolidM is written in JavaScript and available on Github under an MIT license since September 2017. The graphical user interface is realized with WebGME, a web-based generic modeling environment (requires MongoDB). Due to its genericity, the interface is a bit cumbersome; for a user manual see the extended tool description on arXiv [15].

KEVM [16], [17], [18] uses the $\mathbb{K}$ framework to specify the semantics of the EVM formally. The $\mathbb{K}$ framework, developed since 2008 on Github, is a rewriting based system for specifying the formal syntax and semantics of programming languages. From this specification, the tool is able to generate automatically a parser, an interpreter, a model checker, and a deductive program verifier for EVM bytecode. The interpreter passes the standard Ethereum test suite. Moreover, the authors translate the specification of ERC20 tokens to $\mathbb{K}$ and use the generated verifier to analyze the bytecode of three implementations of this token type. KEVM is written in literate programming style as a mixture of markdown syntax and $\mathbb{K}$ specification language. It is available on Github under a non-standard open source license since October 2016.

MAIAN [19], [20] extends the approach of Oyente (see below) by considering also attacks requiring multiple transactions. It executes EVM bytecode symbolically and checks for execution traces indicating that the contract can be self-destructed or drained of Ether from arbitrary addresses, or that it accepts Ether without the function-

ality of a payout. The SMT solver Z3 is used to prune unreachable parts of the search space and to compute transactions that exploit potential vulnerabilities. To discard false positives, the contracts are dynamically analyzed by deploying them on a private blockchain and attacking them with the computed transactions. MAIAN is written in Python and available on Github under an MIT license since March 2018. It uses the Solidity compiler for compiling source code to bytecode and Go-Ethereum for running the private blockchain. Maian is basically a command-line tool, but also provides a simple graphical user interface that requires the graphics library Qt.

Manticore [21], [22] employs symbolic execution to find unique computation paths in EVM (and ELF) binaries. With the help of the SMT solver Z3, it finds inputs that will trigger these computations paths. It records the corresponding execution traces. Regarding the EVM, Manticore compiles Solidity code to bytecode for its analysis, checks the traces for vulnerabilities like reentrancy and reachable selfdestruct operations, and reports them in the context of the source code. Information on the methods and their limitations is scarce. The tool is developed and maintained by the company *Trail of Bits*, and available on Github under an AGPL-3.0 license since February 2017. It can be used from the command-line or via a Python API.

Mythril [23], [24] is a command-line tool in Python for analyzing smart contracts interactively. It executes EVM bytecode symbolically and visualizes the CFG, with the nodes containing disassembled code and the edges being labeled by path formulas. The SMT solver Z3 is used to prune the search space and to compute concrete values for exploiting one potential vulnerability. Checked vulnerabilities are detailed in the online documentation. Mythril is developed and maintained by the company *ConsenSys*, and available on Github under an MIT license since September 2017.

Osiris [25], [26] extends Oyente to detect integer bugs in Solidity smart contracts. It works at bytecode level and constructs a CFG. During symbolic execution, the SMT solver Z3 is queried to determine feasible paths. The tool uses taint analysis (tracking the propagation of data across the control flow of code) to distinguish between benign and malicious overflows. This command-line tool is written in Python and available on Github without explicit license since September 2018.

Oyente [27], [28] is a veteran in the field and has served as starting point for several other projects. It has been regularly used as a reference point. It executes EVM bytecode symbolically and checks for execution traces where transaction order can influence Ether flow, where the result of a computation depends on the timestamp of the block, where exceptions raised by calls are not properly caught, or where a contract can be re-entered multiple times. Unreachable parts of the search space are pruned using the SMT solver Z3. Oyente needs the Solidity compiler for obtaining bytecode, and the disassembler from Go-Ethereum for displaying opcodes in symbolic form. Oyente is written in Python and available on Github under a GPL-3.0 license since January 2016. It is essentially a command-line tool, but offers also a web interface.

Porosity [29], [30] disassembles EVM bytecode, generates a CFG (requires GraphViz for visualization), and decompiles the bytecode into better readable pseudo source code. SSTORE instructions after a CALL are flagged as reentrance vulnerability. This command-line tool is written in C++ and available on Github without explicit license since February 2017, but is no longer maintained.

Rattle [31] improves the readability of EVM bytecode. It reuses Manticore's disassembler, recovers the CFG, and transforms instructions from a stack- to a register-oriented view. The output is presented graphically using GraphViz. This command-line tool is written in Python3, with little documentation. It is developed and maintained by the company *Trail of Bits*, and available on Github under an AGPL-3.0 license since August 2018.

Remix-IDE [32], [33] is an IDE for developing Solidity contracts in a web browser. During compilation it reports security issues, indicating where in the code they occurred. The warnings include implicit visibility, unchecked return values, implicit typing, deprecated constructs, and address checksum. The static analysis is only lightweight and includes some control flow analysis. This tool is available on Github under an MIT license since April 2016, with ample documentation (not for the analyzer, though).

Securify [34], [35] takes EVM bytecode and security properties as inputs. The tool decompiles the stack-oriented bytecode into an assignment-based form and represents the code as DataLog facts. Then it derives further facts that describe the control and data flow in an abstract form. A Security property consists of compliance and violation patterns over-approximating both, satisfaction and non-satisfaction of this property. The patterns are coded as DataLog rules that can be checked against the facts using *Soufflé*. This approach guarantees that if a pattern is detected, the code definitely possesses/violates the corresponding security property. The tool is written in Java and available on Github under an Apache-2.0 license since September 2018. Additionally, a closed source version can be accessed through the website of the company *ChainSecurity* [36]. There is no indication as to the difference between the two versions.

SmartCheck [37], [38] flags potential vulnerabilities in Solidity contracts by searching for specific syntactic patterns in the source code. To this aim, it converts the code into an XML syntax tree. The vulnerabilities are specified as XQuery path expressions that are used to search the patterns in the XML tree. The tool is written in Java and comes in two versions: A command-line version is available on Github under a GPL-3.0 license since May 2017, accompanying the original academic paper. The most recent version of the tool with about twice as many patterns is closed source, and can be accessed via

the website of the company *SmartCheck* [39], which also contains a list of the security issues.

Solgraph [40] visualizes the call flow in Solidity contracts to support users in their analysis. It reads Solidity code and produces a graph, with the nodes representing functions and the directed edges representing function calls. The colors of the nodes mark properties like 'contains send to external address' or 'payable'. This command-line tool is written in JavaScript and is available on Github under an ISC license since July 2016. It uses the Solidity compiler for AST generation and GraphViz for displaying the graph. Documentation is minimal but sufficient.

SolMet [41], [42] is a metric calculator for Solidity code. It uses a parser to generate an AST, on which it computes various software metrics like number of functions, McCabe's style complexity, and depth of nesting levels. This command-line tool is written in Java and is available on Github without license indication since February 2018. Documentation is minimal.

Vandal [43], [44] disassembles and decompiles EVM bytecode into a register-oriented intermediate representation and constructs a CFG. The CFG can be displayed as an interactive HTML page, where clicking on a node of the graph makes the corresponding code appear in a separate box. The intermediate representation can be interpreted abstractly by translating it to Horn clauses and feeding these, together with the specification of vulnerabilities, into the DataLog reasoner *Soufflé*. Vandal is a command-line tool written in Python and is available on Github under a BSD 3-Clause license since August 2016.

B. Tools Not Publicly Available

Here we present tools that have not been made available to the pulic and therefore cannot be installed and tested. Information is taken from publications only.

Ether⋆(s-gram) [45] is a tool for semantic-aware security auditing of Solidity smart contracts by predicting potential vulnerabilities. The authors employ "N-gram language modeling and lightweight static semantic labeling, which can learn statistical regularities." During the learning phase, the tool Oyente was used to determine the vulnerability status of the contracts in the corpus. Ether⋆ is intended to be used as a pre-filter to more resource-consuming tools like ReGuard (see below). Only evaluation data is publicly available [46].

Gasper [47] is a tool for automatically locating gascostly patterns in bytecode. It relies on Oyente for constructing the CFG and for symbolic execution, and searches for dead code and loops containing expensive operations. The tool has not been disclosed in any form.

ReGuard [48] is a tool by the company *Chieftin Lab* for detecting reentrancy vulnerabilities. A web interface allows the user to enter the contract either as Solidity or as bytecode, which is translated to a C++ program via an intermediate representation (AST for Solidity code, CFG for bytecode). The tool runs the C++ program on transactions randomly generated by a fuzzing engine and checks the execution traces for reentrant function calls. The tool has not been disclosed in any form.

SASC [49] is a tool by the company *Fujitsu*. It extends Oyente by rules for additional risks. To present the vulnerabilities detected on bytecode level in the context of the Solidity source code, SASC constructs a call graph with information on events, modifiers, and variables, and correlates it with the instructions on bytecode level. In their publication from March 2018, the authors announce that SASC will become available on Github 'soon'; as of October 2018, it is not yet available.

sCompile [50] takes the bytecode of a contract, constructs a CFG, determines all computation paths involving any flow of Ether, picks those that match patterns characteristic of certain vulnerabilities, ranks them heuristically according to relevance, and finally applies symbolic execution (using the SMT-solver Z3), before presenting the result to the user for manual inspection. The authors plan to make the tool available online, but have not yet revealed the link.

teEther [51] is a tool for automatically creating and verifying exploits for smart contracts given as bytecode. It concentrates on vulnerabilities that cause a payout to arbitrary addresses. After reconstructing the CFG, teEther generates critical paths and uses the SMT-solver Z3 to prune the search space and to compute multi-transactional exploits. To exclude false positives, the exploits are tested on a private blockchain. According to the authors, the tool will be made available on Github by April 2019.

Zeus [52] is a tool developed by *IBM Research India*. It takes Solidity code and a so-called policy as inputs and checks whether the code meets the safety property expressed in the policy. The policy has to be specified by the user. ZEUS makes extensive use of the LLVM compiler infrastructure. Solidity code is translated to LLVM bitcode, which subsequently is instrumented with assertions corresponding to the policy. Then the LLVM code is translated to constrained Horn clauses that are checked with an SMT solver. Only analysis results are publicly available [53], but not the tool itself.

C. Tools Not Considered

We did not include *DappGuard* and *Dr. Y's Ethereum Contract Analyzer* (github.com/pirapira/dry-analyzer). The former does not exist (cf. section I), while the latter is unfinished and has not been changed since July 2017.

IV. Comparison of Tools

In this section, we compare the tools with regard to the following aspects: First, we look at the methods that all tools employ for their analyses. This is followed by implementation details of the available tools. Next, we give an overview of quantitative comparisons, either conducted independently or by the authors of the tools. Finally, we

TABLE I
OVERVIEW OF ALL TOOLS INDICATING PURPOSE, CODE LEVEL, TYPE, PRE-PROCESSING, AND METHODS OF ANALYSIS

	Purpose				Level		Type		Code transformation						Analysis method					
Tool	Security issues	Exploits	Formal guarantees	Bulk analysis	Bytecode	Solidity code	Static analysis	Dynamic analysis	Contextualization	Disassembly	Control flow graph	Call graph	AST analysis	Decompilation	Code instrumentation	Symbolic execution	Constraint solving	Abstract interpretation	Horn logic	Model checking
contractLarva	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
E-EVM	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Erays	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
EthIR	✗	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✗
EtherTrust	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗
FSolidM	✗	✗	✗	✗	form.spec		✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
KEVM	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
MAIAN	✓	✓+	✗	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Manticore	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
Mythril	✓	✓	✗	✗	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Osiris	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Oyente	✓	✗	✗	✓	✓+	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
Porosity	✓	✗	✗	✗	✓+	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Rattle	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗
Remix-IDE	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Securify*	✓	✗	✓	✓	✓+	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✗
SmartCheck*	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Solgraph	✓	✗	✗	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
SolMet	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Vandal	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗
Ether⋆	✓	✗	✗	✓	✓+	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Gasper	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
ReGuard	✓	✗	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SASC	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗
sCompile	✓	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗
teEther	✓	✓+	✗	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗
Zeus	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗

LEGEND. *Tool:* * indicates that we consider the academic version instead of the enhanced company version of the tool, due to availability. *Security issues:* detection of vulnerabilities and potential security problems. *Exploit*: generates exploits, with ✓+indicating multi-transactional exploits. *Formal guarantees:* proves that a contract has a certain property. *Bulk analysis:* suitable for the analysis of large sets of contracts. *Level* of code at which analysis is performed; ✓+ indicates analysis of bytecode including information from the corresponding Solidity code such as the ABI; 'form.spec' means formal specification, from which Solidity code is generated.

contrast the security issues that the tools address. The data for this comparison was gathered in October 2018.

A. Purpose and Methods

Table I contrasts the tools regarding their purpose and methods; for a brief explanation of the methods mentioned therein see section II.

Purpose. Most tools concentrate on security issues, with 18 detecting the presence of vulnerabilities and four (EtherTrust, KEVM, Securify, Zeus) proving their absence. Four tools (MAIAN, Manticore, Mythril, teEther) are able to construct exploits, MAIAN and teEther even such requiring multiple transactions.

Of the remaining tools, E-EVM, Erays, and Rattle disassemble and decompile bytecode for human inspection without automated analysis. ContractLarva adds code to Solidity contracts to check their behavior at runtime, while FSolidM generates Solidity code based on a formal specification by the user. EthIR transforms bytecode such that it can serve as input to general static analyzers, whereas SolMet computes code metrics of Solidity contracts.

About half of the tools are suited for bulk analysis of contracts, e.g. for analyzing the contracts already deployed on the chain. The other half is intended for developing or analyzing individual contracts with human interaction.

Level of abstraction. Most tools start their analysis with the bytecode of contracts. This is due to Solidity lacking formal semantics and changing its behavior between different compiler versions. Nine of these tools also accept Solidity code, but only use the ABI or remember the Solidity code for putting vulnerabilities into context.

FSolidM is exceptional as it works with an abstract

specification of the contract, which the user defines as a finite-state machine.

Methods. Virtually all tools use static analysis, while four tools perform dynamic analyses: ContractLarva checks contract behavior during runtime; MAIAN und teEther try the exploits on the deployed contracts to exclude false positives; ReGuard uses fuzzing techniques after having transformed the contracts.

Tools working on Solidity code typically start by parsing the contract to obtain its AST, and continue analysis from there.

Tools starting from bytecode disassemble the bytecode, generate a CFG, and sometimes decompile it to obtain an intermediate representation. When checking for vulnerabilities, the tools use symbolic execution and an SMT-solver like Z3. Due to the complexity of contracts, it is not possible to cover all computation paths, which means that such tools can detect vulnerabilities, but cannot prove their absence.

Tools that target such security guarantees simplify the code in a sound manner and use abstract interpretation to cover all computation paths. This is typically done by translating the simplified contract to Horn clauses and employing a tool like Souflé to show that some security property holds. In case the tool succeeds, the contract is free from the vulnerabilities covered by the property.

B. Implementation Details

To evaluate the continuity and complexity of the publicly available tools, we collected metric data from their Github repositories. EtherTrust is available via a static webpage and does not offer the same data. For Securify and SmartCheck we examine the open 'academic' version on Github as there is no data available for the closed 'company' version.

For assessing the support and longevity of the tools we consider the number of commits, the date of the first commit, the active months (difference between last and first commit), and the number of contributors. Moreover, we mark a tool as *publication tool*, if the last noteworthy commit coincides with a date relevant to the conference where the accompanying paper was published. Regarding the complexity of the tools, we counted the lines of code (disregarding code copied from other projects or generated automatically) and determined the programming languages used. The data is compiled in table II.

The table suggests that academic tools tend to be 'publication tools', developed as a proof-of-concept and published for the sake of the conference. Functionality, usability and documentation are minimal, while the future of the tools remains uncertain. There are some notable exceptions, though. The most impressive one is KEVM with by far the biggest team among the academic tools and a continuous development history. It is followed by the tools Vandal, Ethir, and FSolidM, which are also being developed over an extended period of time. The more recent academic tools Erays, Osiris, and SolMet have yet to prove their continuity.

TABLE II
IMPLEMENTATION DETAILS OF AVAILABLE TOOLS

Tool	Commits	First commit mm/yy	Active months	Publication tool	Contributors	Affiliation	k loc	Language
contractL.	14	12/17	7	✓	1	ac	2.8	hs
E-EVM	2	1/18	1	✓	1	ac	2.0	py
Erays	6	8/18	3	*	3	ac	5.6	py
EthIR	394	2/18	9		1	ac	7.9	py
EtherTr.	1	5/18	na	✓	na	ac	13.0	java
FSolidM	183	9/17	13		4	ac	35.0	js
KEVM	1581	10/16	25		21	ac	23.0	$\mathbb{K}$,py
MAIAN	14	3/18	2	✓	2	ac	3.1	py
Manticore	597	2/17	21		56	co	37.0	py
Mythril	1988	9/17	14		42	co	9.0	py
Osiris	4	9/18	1	*	1	ac	1.2	py
Oyente	799	1/16	31		22	cm	6.6	py
Porosity	94	2/17	12		10	co	4.2	cpp
Rattle	23	8/18	2		2	co	2.6	py
Remix-I	4972	11/14	48		63	cm	17.0	js
Securify*	21	9/18	2	✓	4	ac	13.0	java
SmartCh.*	161	5/17	10	✓	4	ac	2.1	java
Solgraph	68	7/16	26		3	co	0.1	js
SolMet	9	2/18	7	*	1	ac	0.6	java
Vandal	811	8/16	22		6	ac	7.3	py

LEGEND. *Tool:* * marks the academic version if there is a private company version. *Active months:* number of months between first and last commit. *Publication tool:* ✓ means that the last noteworthy commit coincides with some important conference date; * means that the tool has been published only recently; see text for explanation. *Affiliation* of tool authors: ac(ademic), co(mpany), cm for community. *Language:* hs=Haskel, py=Python, js=JavaScript, java=Java, cpp=C++, $\mathbb{K}$=specification language of the $\mathbb{K}$ framework.

The community tools Remix-IDE and Oyente show a longstanding development and a large support team.

The big team sizes of the community tools are only paralleled by the company tools Manticore and Mythril. As for the other company tools, Porosity is marked unmaintained on Github, while Solgraph looks maintained. Rattle is too new (first commit in August 2018) to be judged.

Regarding the implementation languages, Python is the most popular one with 11 instances. Four tools use Java, three JavaScript, one each use Haskell and C++. KEVM leverages the $\mathbb{K}$ framework.

C. Quantitative Comparisons

Next, we look at quantitative reviews of the tools. These are grouped into reviews by authors, who evaluate their own tool against others, and 'independent' ones by authors without a tool of their own involved in the comparison.

The three independent reviews are contrasted in table III. In [54], the tools Oyente, Mythril, Securify, and SmartCheck are compared with regard to their effectiveness and accuracy in detecting known vulnerabilities.

TABLE III
INDEPENDENT TOOL COMPARISONS

	Remix	Porosity	SmartCheck	Securify	Mythril
Oyente	[56]	[55]	[54], [56]	[54], [56]	[54], [55]
Mythril		[55]	[54]	[54]	
Securify	[56]		[54], [56]		
Smartcheck	[56]				

TABLE IV
COMPARISON OF OWN TOOLS WITH OTHERS

Tool \ to tool	Oyente	Mythril	Securify	Remix	Zeus	MAIAN	EthIR	Rattle
Vandal	[44]	[44]					[44]	[44]
SmartCh.	[38]		[38]	[38]				
Securify	[35]	[35]						
sCompile	[50]					[50]		
teEther	[51]				[51]			
SASC	[49]							
ReGuard	[48]							
Zeus	[52]							
Osiris					[26]			

In [55], the tools Oyente, Mythril, and Porosity are compared by contrasting claimed and successfully detected security issues. Regarding vulnerability checking capabilities, [56] compares the tools Oyente, Remix, Securify, SmartCheck in detail.

The following nine references provide a quantitative comparison of their own tool with at least one other tool: [44] compares Vandal to Oyente, EthIR, Rattle, and Mythril. [38] compares SmartCheck to Oyente, Remix, and Securify. [35] compares Securify to Oyente and Mythril. [50] provides a comparison of sCompile with Oyente and MAIAN regarding execution time. [51] compares teEther to Oyente and Zeus. [49] compares SASC to Oyente. [48] compares ReGuard to Oyente. [52] compare their tool Zeus to Oyente. [26] compares Osiris to Zeus because the latter can also detect integer bugs.

Table IV shows that most tools are compared to Oyente. This is not surprising as Oyente was the first symbolic execution tool published.

As can be seen from tables III and IV:

- Six tools (Mythril, Remix, Porosity, MAIAN, EthIR, and Rattle) do not compare themselves to any other tool.
- Another six tools (Vandal, sCompile, teEther, SASC, ReGuard, Osiris) have not yet been used for comparison. One reason might be that four of them are not publicly available, while Osiris and Vandal have been published only recently.
- The tools Oyente, Mythril, Securify, and Remix appear in both tables as candidates for comparison.

D. Detection of Vulnerabilties

In this section, we examine the available tools and compare them with regard to the security issues they detect. The tools contractLarva, E-EVM, Erays, EthIR, FSolidM, KEVM, Rattle, and SolMet do not claim to find security issues. EtherTrust takes a different approach to recognizing security issues. It proves the two properties 'single entrancy' and 'independence from the transaction environment' for bytecode.

The remaining 11 tools analyze several known security issues, which we contrast in table V . The columns are based on the original categorization by [57]. To accommodate for the aspects that the tools actually examine, we adjusted the original categories: 'Immutable bugs' and 'Type casts' have been omitted as this is not checked by any tool. A few categories have been renamed, several new ones have been added. We distilled the following security issues that the tools analyze:

TOD (Transaction-ordering dependence, also called front running, race condition): the ordering of transactions cannot be relied on in a contract.

Random number (also called 'nothing is secret'): has to be handled with care on a blockchain.

Timestamp dependence: as the timestamp can be manipulated, its usage may be unsafe.

Unpredictable state: calling a library may effect in an unforeseen change of the state of the calling contract.

Callstack depth (until Oct 18, 2016): an external call can fail because it exceeds the maximum call depth.

Lost Ether: Ether transferred to an orphaned address cannot be retrieved.

Reentrancy: A contract transfers Ether to another contract and hands control over to it, and is called back from this contract before the transaction has been completed.

Unchecked call (incl. gasless send, bad exception handling): the low level functions call, callcode, delegatecall and send are potentially risky and should be avoided. They do not revert on failure, but return false and continue.

tx.origin: its use for authorization is highly discouraged.

Blockhash: can be manipulated to some degree.

Send: can fail.

Selfdestruct: should be checked for proper authorization.

Visibility: the unintended visibility of functions and state variables may cause a vulnerability (e.g. leaking secrets, function exposure).

Unchecked math (incl. integer overflow): may incur a vulnerability.

Costly patterns (gas): can cause a transaction to fail.

Bad coding pattern: may incur a vulnerability.

Deprecated: disapproved Solidity elements should no longer be used.

Other: the tool checks other issues as well.

TABLE V
SECURITY ISSUES CHECKED BY AVAILABLE TOOLS

	Blockchain				EVM		Solidity											
Tool	TOD	Random number	Timestamp	Unpredictable state	Callstack depth	Lost Ether	Reentrancy	Unchecked call	tx.origin	Blockhash	send	selfdestruct	Visibility	Unchecked math	Costly pattern	Bad coding pattern	Deprecated	Other
MAIAN	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Manticore	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	✓
Mythril	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓
Osiris	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Oyente	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Porosity	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Remix-IDE	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Securify*	✓	✗	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓
SmartCheck*	✗	✗	✓	✓	✗	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓
Solgraph	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✓
Vandal	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✓	✗	✗	✗	✗	✗	✓

The first four columns refer to blockchain issues. Three tools (MAIAN, Solgraph, Vandal) omit them completely. None of the tools analyzes them all. A similar situation is encountered with regard to the two columns, which refer to EVM issues. Five tools (Manticore, Osiris, Remix-IDE, Solgraph, Vandal) skip them. No tool checks them all.

The picture for the Solidity issues is quite diverse. Some tools are focused on only a few issues. Osiris is specialized to integer bugs, while Porosity only checks reentrancy. MAIAN analyzes three specific vulnerabilities. Oyente targets four vulnerabilities. The remaining tools target five or more issues. Five tools try to capture a whole range of vulnerabilities and bad practices: Manticore, Mythril, Remix-IDE, Securify and SmartCheck.

The most popular vulnerability recognized seems to be 'reentrancy' (checked by eight tools), followed by 'unchecked calls' (checked by six tools).

Table V does not indicate the effectiveness of the tools in detecting security issues.

V. CONCLUSIONS

To better assess the detection quality of tools, standardized benchmarks are desirable. On Solidity level, a widely used test set is the collection of verified contracts on Etherscan (mainly because it is available). Furhermore, there are collections of contracts with 'known vulnerabilities' like *Ethernaut* [58] or *Not so Smart Contracts* [59]. For bytecode analyses, evaluations are usually based on the contracts deployed on the main chain. These test sets are readily obtainable but sub-optimal. On the one hand, the collections are unbalanced, as certain types of contracts (and their vulnerabilities) prevail. On the other hand, the sheer mass of contracts renders it difficult to establish a reliable ground truth, i.e., the correct status of each contract w.r.t. to each vulnerability. We need a benchmark suite of carefully selected or crafted contracts being vulnerable or immune against each of the attacks. The common practice of using a tool (e.g. Oyente) as an oracle is at least questionable.

We would like the scientific community to discourage publications with undisclosed tool implementations as the claims cannot be verified. Moreover, in the area of untrusted computing, open source is key to build trust. According to our findings, tools will persist when they are openly developed on a broad basis.

When starting to develop a new tool, we suggest to reuse components like disassembler, decompiler, or parser from the repositories of the *Ethereum Foundation* (github.com/ethereum), of *Parity Technologies* (github.com/paritytech), of *ConsenSys* (github.com/ConsenSys), of *Trail of Bits* (github.com/trailofbits), and of the more advanced tools presented here.

We conclude our survey by highlighting five tools that we found particularly inspiring. *FSolidM* pursues the noteworthy approach of generating source code from a state-oriented specification. Regarding formal verification, *KEVM* is the clear favorite due to its maturity, with the drawback that its use requires expertise. *Securify* is the most advanced tool regarding formal guarantees. *MAIAN* is founded on a precise definition of the vulnerabilities to be detected and, compared to the other tools, goes the extra mile to discard false positives by testing the exploits. Finally, *Mythril* is a prime example of a tool for analyzing contracts interactively, with an eye on usability.

REFERENCES

[1] T. Cook, A. Latham, and J. H. Lee, "Dappguard: Active monitoring and defense for solidity smart contracts," MIT, student project, 2017, https://courses.csail.mit.edu/6.857/2017/project/23.pdf.

[2] G. Pace, "contractLarva," Oct 2018, https://github.com/gordonpace/contractLarva.

[3] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: Contractlarva and open challenges beyond," in *18th Int. Conf. on Runtime Verification (RV'18)*, ser. LNCS, vol. 11237. Springer, 2018, pp. 113–137, https://doi.org/10.1007/978-3-030-03769-7_8.

[4] pisocrob, "E-EVM," Oct 2018, https://github.com/pisocrob/E-EVM.

[5] R. Norvill, B. B. F. Pontiveros, R. State, and A. J. Cullen, "Visual emulation for Ethereum's virtual machine," in *Network Operations and Management Symposium (NOMS'18)*. IEEE, 2018, pp. 1–4, https://doi.org/10.1109/NOMS.2018.8406332.

[6] teamnsrg, "Erays," Oct 2018, https://github.com/teamnsrg/erays.

[7] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: Reverse engineering Ethereum's opaque smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1371–1385, https://www.usenix.org/conference/usenixsecurity18/presentation/zhou.

[8] P. Gordillo, "EthIR," Oct 2018, https://github.com/costa-group/EthIR.

[9] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," *arXiv:1805.07208*, 2018.

[10] I. Grishchenko, "EtherTrust," Oct 2018, https://www.netidee.at/ethertrust.

[11] I. Grishchenko, M. Maffei, and C. Schneidewind, "EtherTrust: Sound static analysis of ethereum bytecode," Technische Universität Wien, Tech. Rep., 2018.

[12] A. Mavridou and A. Laszka, "FSolidM," Oct 2018, https://github.com/anmavrid/smart-contracts.

[13] ——, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *Int. Conf. on Principles of Security and Trust*. Springer, 2018, pp. 270–277.

[14] ——, "Designing secure ethereum smart contracts: A finite state machine based approach," in *22nd Int. Conf. on Financial Cryptography and Data Security (FC'18)*. Springer, 2018.

[15] ——, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," *arXiv:1802.09949*, 2018.

[16] kframework.org, "KEVM," Oct 2018, https://github.com/kframework/evm-semantics.

[17] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 204–217.

[18] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Rosu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'18)*, 2018, pp. 18–21.

[19] MAIAN-tool, "MAIAN," Oct 2018, https://github.com/MAIAN-tool/MAIAN.

[20] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv:1802.06038*, 2018.

[21] Trail of Bits, "Manticore: Symbolic Execution for Humans," Oct 2018, https://github.com/trailofbits/manticore.

[22] F. Manzano and J. Feist, "Automatic bug finding for the blockchain," 2017, https://tinyurl.com/yby396gd.

[23] ConsenSys, "Mythril," Oct 2018, https://github.com/ConsenSys/mythril-classic.

[24] B. Mueller, "Smashing smart contracts," in *9th HITB Security Conference*, 2018, https://tinyurl.com/y827tk72.

[25] C. Ferreira, "Osiris," Oct 2018, https://github.com/christoftorres/Osiris.

[26] C. Ferreira Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *34th Annual Computer Security Applications Conference (ACSAC'18)*, 2018.

[27] melon.fund, "Oyente," Oct 2018, https://github.com/melonproject/oyente.

[28] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016, pp. 254–269.

[29] Comae Technologies, "Porosity," Oct 2018, https://github.com/comaeio/porosity.

[30] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," DEF CON 25, Tech. Rep., 2017, https://tinyurl.com/y9kb47dr.

[31] Trail of Bits, "Rattle," Oct 2018, https://github.com/trailofbits/rattle.

[32] Ethereum Foundation, "Remix-IDE," Oct 2018, https://github.com/ethereum/remix-ide.

[33] ——, "Remix Documentation," Oct 2018, https://remix.readthedocs.io/en/latest/index.html.

[34] SRI Lab, ETH Zurich, "Securify: public version," Oct 2018, https://github.com/eth-sri/securify.

[35] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *arXiv:1806.01143*, 2018.

[36] ChainSecurity, "Securify," Oct 2018, https://securify.chainsecurity.com/.

[37] SmartDec, "SmartCheck: academic version," Oct 2018, https://github.com/smartdec/smartcheck.

[38] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18. ACM, 2018, pp. 9–16.

[39] SmartDec, "SmartCheck," Oct 2018, https://tool.smartdec.net/.

[40] R. Revere, "Solgraph," Oct 2018, https://github.com/raineorshine/solgraph.

[41] P. Hegedus, "SolMet," Oct 2018, https://github.com/chicxurug/SolMet-Solidity-parser.

[42] ——, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," in *IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018, pp. 35–39.

[43] Smart Contract Research at USYD, "Vandal," Oct 2018, https://github.com/usyd-blockchain/vandal.

[44] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv:1809.03981*, 2018.

[45] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proc. 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 814–819.

[46] H. Liu, "s-gram," Oct 2018, https://github.com/njaliu/sgram-artifact.

[47] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 442–446.

[48] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 65–68.

[49] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*. IEEE, 2018, pp. 1–5.

[50] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," *arXiv:1808.00624*, 2018.

[51] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018, pp. 1317–1333.

[52] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Network and Distributed Systems Security (NDSS) Symposium 2018*. NDSS, 2018.

[53] S. Kalra, "Zeus data," Oct 2018, https://tinyurl.com/y7z7ha6q.

[54] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," pp. 103–113, 2018.

[55] R. Fontein, "Comparison of static analysis tooling for smart contracts on the evm," in *28th Twente Student conference on IT*, 2018.

[56] A. Dika, "Ethereum smart contracts: Security vulnerabilities and security tools," Master's thesis, NTNU, 2017.

[57] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[58] OpenZeppelin, "Ethernaut – Solidity security challenges," https://github.com/OpenZeppelin/ethernaut, accessed 2018-08-07.

[59] Trail of Bits, "Not So Smart Contracts," Oct 2018, https://github.com/trailofbits/not-so-smart-contracts.