Delft University of Technology

Evolution of automated weakness detection in Ethereum bytecode
a comprehensive study

di Angelo, Monika; Durieux, Thomas; Ferreira, João F.; Salzer, Gernot

DOI
10.1007/s10664-023-10414-8

Publication date
2024

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
di Angelo, M., Durieux, T., Ferreira, J. F., & Salzer, G. (2024). Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study. *Empirical Software Engineering*, *29*(2), Article 41. https://doi.org/10.1007/s10664-023-10414-8

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Empirical Software Engineering
https://doi.org/10.1007/s10664-023-10414-8

Evolution of automated weakness detection in Ethereum bytecode: a comprehensive study

Monika di Angelo[1] · **Thomas Durieux**[2] · **João F. Ferreira**[3] · **Gernot Salzer**[1]

Accepted: 18 October 2023
© The Author(s) 2024

Abstract

Blockchain programs (also known as smart contracts) manage valuable assets like cryptocurrencies and tokens, and implement protocols in domains like decentralized finance (DeFi) and supply-chain management. These types of applications require a high level of security that is hard to achieve due to the transparency of public blockchains. Numerous tools support developers and auditors in the task of detecting weaknesses. As a young technology, blockchains and utilities evolve fast, making it challenging for tools and developers to keep up with the pace. In this work, we study the robustness of code analysis tools and the evolution of weakness detection on a dataset representing six years of blockchain activity. We focus on Ethereum as the crypto ecosystem with the largest number of developers and deployed programs. We investigate the behavior of single tools as well as the agreement of several tools addressing similar weaknesses. Our study is the first that is based on the entire body of deployed bytecode on Ethereum's main chain. We achieve this coverage by considering bytecodes as equivalent if they share the same skeleton. The skeleton of a bytecode is obtained by omitting functionally irrelevant parts. This reduces the 48 million contracts deployed on Ethereum up to January 2022 to 248 328 contracts with distinct skeletons. For bulk execution, we utilize the open-source framework SmartBugs that facilitates the analysis of Solidity smart contracts, and enhance it to accept also bytecode as the only input. Moreover, we integrate six further tools for bytecode analysis. The execution of the 12 tools included in our study on the dataset took 30 CPU years. While the tools report a total of 1 307 486 potential weaknesses, we observe a decrease in reported weaknesses over time, as well as a degradation of tools to varying degrees.

Keywords Bytecode · Blockchain · Debugging · Detection tools · Ethereum · EVM · Program analysis · Reproducible Bugs · Smart contracts · Vulnerability

Communicated by: Eric Bodden

✉ Monika di Angelo
monika.di.angelo@tuwien.ac.at

Extended author information available on the last page of the article

Published online: 02 February 2024

1 Introduction

Smart contracts are event-driven programs running on the nodes of decentralized networks known as *blockchains*. Specific transactions, once included in the blockchain, trigger the execution of these blockchain programs. Every node executes the code locally within a virtual machine and updates its state of the blockchain. The computations are deterministic, ensuring that all nodes arrive at the same state. The flexibility of smart contracts and the unique properties of blockchains, most notably decentralization and immutability, gave rise to innovative applications in areas like decentralized finance and supply chain management. Their potential has led to ecosystems with large numbers of start-ups and market caps of hundreds of billions of USD.

Against this background, errors in smart contracts can lead, and have led, to costly disruptions and losses. Early on, academia and industry focused on methods and tools for developing *secure* smart contracts. In a survey on automated vulnerability detection conducted in mid-2021, Rameder et al. (2022) identified 140 tools for Ethereum, the major smart contract platform. The sheer number makes it hard to decide which tools may be suited for the task at hand, and calls for regular tool evaluations and comparisons.

In this paper, we present a comprehensive evaluation of 12 tools for vulnerability detection on the Ethereum main chain. The goal of this study is to analyze how typical tools behave within the Ethereum ecosystem. We focus on the evolution of tools and their findings to identify common patterns and trends. The results of our study can be utilized to inform developers about the state of the art in automated vulnerability detection and to guide researchers in the development of new tools. Additionally, it provides an overview of the reliability of the selected tools and whether they should be included in future studies.

Our work differs from previous studies in several aspects. First, we analyze the *temporal evolution of weakness detection*, focusing on the robustness of tools over time (rather than assessing their detection capabilities against a set of contracts).

Second, we aim at a *complete coverage of the Ethereum main chain*, which is a formidable endeavor in light of 48 million deployments of smart contracts (up to Jan 2022). This enables us to investigate the evolution of weakness detection over a period of more than six years. We select one contract per skeleton of bytecode (cf. Section 3), which reduces the number of objects to analyze to 248 328.

Third, we concentrate on the *runtime bytecode* as input to the tools. Surveys usually evaluate tools on benchmarks of Solidity source code (cf. Section 12), and hence omit tools analyzing bytecode only. Moreover, for many contracts on the blockchain, the source code is not available. By choosing runtime bytecode as the least denominator, we can include tools rarely considered, and we are able to analyze all smart contracts deployed so far.

Finally, to perform our study, we extend SmartBugs (Ferreira et al., 2020), a framework for executing analysis tools in a unified manner. Integrating new tools into the framework makes them available for future evaluations by others.

With 12 tools, 15 weakness classes, 248 328 runtime bytecodes of smart contracts and an execution time of 30 years, our evaluation is more comprehensive than previous studies. The large number of samples as well as the method of selection allows us to add a unique temporal perspective. In summary, the contributions of this paper are:

- A method for selecting a feasible number of smart contracts that are representative of 48 M blockchain programs deployed on Ethereum in the course of six years.

- A public dataset of 248 328 smart contracts that may serve as the basis of further evaluations. [1]
- An extension of the framework SmartBugs to include 12 tools for vulnerability detection with bytecode-only input. [2]
- Methods for analyzing and visualizing the temporal evolution of tool results and the overlap between tools.
- A portrait of the evolution of tool behavior and weakness detection on six years of blockchain activity.

2 Study design

Our study aims to provide a comprehensive perspective on the evolution of weaknesses in smart contracts, as detected by automated analysis tools. We focus on the Ethereum blockchain, which is the major platform for smart contracts in terms of the number of applications, market cap, attacks, and countermeasures. To address the research questions outlined below, we proceed as follows.

Collecting the Contract Data. The study period covers the 14 million blocks from Ethereum's start on 30 July 2015 up to 13 Jan 2022. During this period, there were 48 million contract deployments. Analyzing the contracts in their entirety is not only infeasible, but wastes resources and introduces biases, as the deployments range from one-of-a-kind contracts to code deployed hundreds of thousands of times. In Section 3, we introduce the *skeleton* of contracts. By grouping contracts with identical skeletons and selecting only one contract per group, we capture the diversity of the Ethereum ecosystem by analyzing just 248 328 contracts. We assess various properties of the data needed later on.

Selecting the Analysis Tools. Performing a large-scale analysis on smart contracts that, in general, are only available as bytecode, restricts the number of applicable tools. In Section 4, we specify the selection criteria and apply them to the 140 tools identified by Rameder et al. (2022). We describe the 12 tools that remain regarding engineering aspects and their approach to contract analysis.

Analyzing the contracts. To execute 12 tools with diverse requirements and I/O formats on 248 328 contracts, we select the execution framework *SmartBugs*. Initially, it contained only half of the needed tools and required Solidity source code as input. We extended SmartBugs to accept bytecode as well and added the other six tools. After a cumulative execution time of 30 years, we obtain three million records, each specifying the result of running a single tool on a specific bytecode. We refer to Section 5 for the details regarding the execution framework.

Mapping the Weaknesses to a Common Taxonomy. The execution data allows us to analyze the detection results per tool. To facilitate the comparison and aggregation of results from multiple tools, we map the tool findings to a common taxonomy that is described in Section 6.

Based on the results of running the analyzers on the bytecodes, we address the following research questions.

RQ1 Abstraction. *How well are skeletons suited as an abstraction of functionally similar bytecode in the context of weakness analysis?* In Section 7, we investigate whether and how the weakness analysis of bytecodes with the same skeleton differs.

[1] Available at https://github.com/gsalzer/skelcodes

[2] Available at https://github.com/smartbugs/smartbugs

RQ2 Weakness Detection. *Which trends in the weakness reports of analysis tools can be identified for the contracts on Ethereum's main chain?* In Section 8, we are interested in the evolution of types and numbers of weaknesses reported for the deployments up to early 2022.

RQ3 Tool Quality. *How do analysis tools change their behavior in a weakness analysis with bytecode input?* In Section 9, we investigate the evolution of tool quality with respect to maintenance aspects, execution time, errors, and failures.

We do not assess the individual performance of the tools, like the rates of true/false positives/negatives, as there is no ground truth that is sufficiently large, consistent and balanced for a conclusive evaluation (di Angelo and Salzer, 2023).

RQ4 Overlap Analysis. *To which extent do the tools agree when addressing similar weaknesses?* In Section 10, we determine the overlap of tools for weaknesses that can be mapped to the same class of the SWC registry.

Discussion. In Section 11, we combine the results of our research questions. For two specific SWC classes, we investigate how the agreement of the tools evolve over time, and provide an explanation. Moreover, we consider the limitations of our study.

Related work. Section 12 gives an overview of studies similar to ours and highlights the differences. Moreover, it compares two execution frameworks and justifies our decision for using SmartBugs.

3 Contract data

This section describes the collection and preparation of the contract data that we are going to analyze. Moreover, we assess some characteristics of the data that will be needed to interpret the results of the analysis tools. We start by clarifying some concepts specific to Ethereum.

3.1 Creation of contracts

Deployment vs. runtime code. To deploy a contract on an Ethereum chain, an external user submits a create transaction, or the Ethereum Virtual Machine (EVM) executes a create operation. The transaction/operation includes the *deployment code*. The code consists of an active part, D, which typically initializes the environment for the new contract. At its end, D returns the pointer to a memory area with the actual *runtime code*, which the EVM then stores at the address of the new contract. The deployment code is free to assemble the runtime code arbitrarily, but typically just copies code following D in the deployment code.

Source code vs. bytecode. The majority of Ethereum contracts are written in Solidity, a programming language inspired by C++. The so-called constructor and any global initializations compile to the active part of the deployment code, D, whereas all other parts of the source file compile to the runtime code proper, R, which is appended to D. After R, the compiler appends *meta-data*, M, which contains a hash identifying the original source code and version information. Changing any character in the Solidity file, including comments and the newline encoding, alters M and leads to superficially different deployment and runtime codes.

To illustrate the role of the different forms of code, consider the program in Listing 1. It shows the Solidity code of a contract `C1` that deploys a contract of type `C2` as part of its own deployment. At runtime, each call to function `f` deploys a contract of type `C3`. From this source code, the compiler generates a bytecode of the form $D_1 R_1 D_3 R_3 M_3 M_1 D_2 R_2 M_2$. During deployment, D_1 creates the contract `C2` by executing $D_2 R_2 M_2$, with the runtime

Table 1 Deployments up to Block 14 M (Jan 2022)

Metric	# Contracts
Deployments	48 262 411
Distinct deployment codes	2 206 793
Distinct runtime codes	514 893
… without meta-data	364 599
… without PUSH arguments	249 076
Skeletons	248 328

code R_2M_2 getting stored at the address of C2. Then D_1 returns $R_1D_3R_3M_3M_1$ as runtime code, which is stored at the address of C1. Later, when the function f of contract C1 is called, $D_3R_3M_3$ gets executed and creates a new contract with runtime code R_3M_3. Note the multiple occurrences of meta-data and their proliferation during deployment and runtime.

Listing 1: Solidity contract creating contracts during deployment (C2) as well as during runtime (C3). The definitions of C2 and C3 have been omitted.

```
contract C1 {
    C2 c = new C2();
    function f() public returns(C3) {
        return new C3();
}   }
```

Skeletons. The skeleton of a contract is obtained by removing meta-data, the arguments of PUSH operations, constructor arguments, and trailing zeros. The rationale is to remove parts that contribute little to the functionality of the contract, with the aim to equate contracts with the same skeleton.

Code family. A *family of codes* is a collection of runtime codes with the same skeleton.

3.2 Data Collection

We strive for a complete coverage of Ethereum's main chain. Therefore, we collect the runtime codes of all contracts (including the self-destructed ones) that were successfully deployed up to block 14M (13 Jan 2022).[3] Table 1 gives an overview of the deployment activities. The 48.3M contract creations involved 2.2M different deployment codes, generating a total of 0.5 M distinct runtime codes. The removal of meta-data reduces the number of distinct codes by 29%, and the removal of PUSH constants by another 22%.

For each family of codes, i.e., for each collection of codes sharing the same skeleton, we pick a single representative and omit the others. For practical purposes, we prefer deployments where Etherscan lists the corresponding source code. The longest-lived family consists of two codes implementing an ERC20 token, deployed 17 333 times over a range of almost 12 million blocks, whereas the most prolific family consists of 20 codes deployed over 12 million times. [4] The size of the families seems to follow a Pareto principle: 84% of the families are singletons (the skeleton is uniquely associated with a single runtime code), 15%

[3] We used an OpenEthereum client, https://github.com/openethereum/openethereum.

[4] The codes of this family, 21 bytes in length, belong to gas token systems. When called from the address mentioned in the code, the contracts self-destruct, leading in earlier versions of Ethereum to a gas refund.

Table 2 Forks on Ethereum's main chain introducing new operations

Name of fork	Activated at	New operations
Homestead	1.150 M	`DELEGATECALL`
Byzantium	4.370 M	`RETURNDATASIZE`, `RETURNDATACOPY`, `REVERT`, `STATICCALL`
Constantinople	7.280 M	`CREATE2`, `EXTCODEHASH`, `SAR`, `SHL`, `SHR`
Istanbul	9.200 M	`CHAINID`, `SELFBALANCE`
London	12.965 M	`BASEFEE`

of the families consist of 2 to 10 codes, whereas at the other end of the spectrum we find a skeleton shared by 16 372 codes. [5]

We obtain a dataset of 248 328 runtime codes with distinct skeletons that represent all deployments up to January 13, 2022. 99.0 % of these codes originate from the Solidity compiler (as determined by characteristic byte sequences), with the source code for 46.5% actually available on Etherscan. *For our temporal analyses, we associate each code with the block number where the first member of the family was deployed.*

Not all bytecodes are proper contracts. In particular in the early days of the main chain, during an attack, a number of large 'contracts' were deployed that served as data repositories for other contracts. For some tools, this leads to a noticeable spike in the error rate around block 2.3 M.

3.3 Forks Introducing New Operations

Over time, Ethereum has seen several updates, so-called forks. Some of them add new operations to the EVM; Table 2 gives an overview of those relevant to our study. When interpreting the results of an analysis tool, we have to relate the age of the tool to these forks in order to understand the effect of new operations.

The actual use of new operations varies. It depends on the integration of the operations into the Solidity compiler, the adoption of new compiler versions by the contract developers, and on the relevance of the operation. Figure 1 shows the use of compiler versions as well as the share of contracts using a particular operation. In the remainder of the section, we first explain the origin of the data and then discuss the details of the figure.

For bytecodes with a source code on Etherscan.io (46.5 %), we take the compiler version from there. For the others, we try to extract it from the embedded meta-data, succeeding for another 22.8% of the bytecodes. Given that 99 % of the bytecodes have been generated by the Solidity compiler, we extend the compiler distribution in each bin of 100 k blocks from the codes with a known version to the remaining ones. For the stackplot in the background of Fig. 1, this amounts to considering the share of 69.3 % bytecodes with version information as 100% of the data.

We group the 100+ compiler versions into 10 ranges, such that from each range to the next, at least one additional operation from Table 2 is supported, either by Solidity offering a new language element related to the operation, or by the compiler using the new operation internally to generate better bytecode.

[5] These codes are proxy contracts of 45 bytes that forward any incoming call to a fixed address. The 16 372 codes only differ with respect to this hard-coded address, occurring as the argument of a `PUSH` operation.

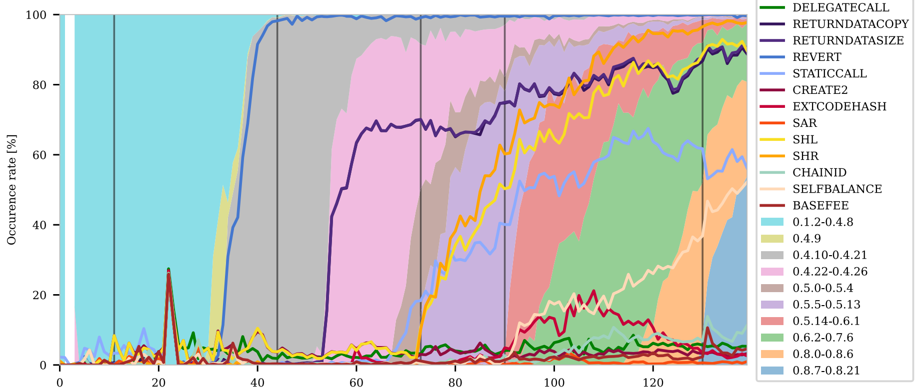

Fig. 1 The use of new EVM operations as well as the adoption of new compiler versions over time. In the foreground, each line shows the percentage of bytecodes containing a particular operation, per bin of 100 k blocks. The background shows the distribution of compiler versions for each bin. DELEGATECALL appears from compiler version 0.4.9 onwards, REVERT from 0.4.10, RETURNDATACOPY and RETURNDATASIZE from 0.4.22, STATICCALL from 0.5.0, SAR, SHL and SHR from 0.5.5, SELFBALANCE from 0.5.14, CREATE2 from 0.6.2, CHAINID from 0.8.0, and BASEFEE from 0.8.7 onwards. EXTCODEHASH is only available via assembly code

It is inherently difficult to determine the parts of a bytecode that are reachable and may get executed. To obtain a list of the operations in a bytecode, we identify the first code block and extract its operations linearly. This yields an overapproximation, as data embedded in the code may be mistaken as operations.

In Fig. 1, the effect of this overapproximation is visible up to block 3 M. Except for DELEGATECALL, none of the new operations has been added to the EVM yet, but the plots show their occurrence at a rate of up to 8 % nonetheless. The spike at block 2.3 M, with all operations seemingly occurring simultaneously at a rate of 28 %, is an artefact caused by a large number of 'contracts' consisting entirely of data (the by-product of an attack). The first real phenomenon to observe is the REVERT operation (medium blue line), introduced by compiler version 0.4.10 (gray area). The compiler starts to use REVERT for exception handling shortly after 3 M, even though the operation is added to the EVM only at fork 4.37 M. Before the fork, REVERT causes an EVM exception just like the invalid opcode used for this purpose until then, but performs a more refined error handling after the fork.

The other operations of fork 4.37 M are adopted by the Solidity compiler with some delay. From version 0.4.22 onwards, the pair of RETURNDATA operations (lines in two shades of violet, with the plots almost coinciding) is used for most calls and eventually occurs at a rate beyond 80 %. STATICCALL (light blue line) gets in use even later, rising to a rate of 60 % towards the end of the timeline.

The shift operations SHL (yellow line) and SHR (orange line) are of particular importance. At the end of the study period, they occur in almost every contract. As the function dispatcher at the start of a contract now uses SHR, this operation gets executed with virtually every invocation of a contract. Of the other operations, only SELFBALANCE and EXTCODEHASH are used to some extent, while DELEGETECALL, CREATE2, SAR, CHAINID and BASEFEE play a lesser role. [6]

[6] The low number of contracts containing DELEGATECALL may seem surprising, given that this operation is essential for hundreds of thousands of proxy contracts. But as these contracts show little variety (few skeletons), they are represented by only a small number of contracts in our dataset.

Table 3 Tools Selected for the Study

Tool	Reference	URL	Version
Conkas		github.com/nveloso/conkas	6aee09
Ethainter	Brent et al. (2020)	zenodo.org/record/3760403	2b26bf
eThor	Schneidewind et al. (2020)	secpriv.wien/ethor/	6405e0
MadMax	Grech et al. (2018)	github.com/nevillegrech/MadMax	6e9a6e9
Maian	Nikolić et al. (2018)	github.com/ivicanikolicsg/MAIAN	4bab09
Mythril	Mueller (2018)	github.com/ConsenSys/mythril	0.22.32
Osiris	Ferreira Torres et al. (2018)	github.com/christoftorres/Osiris	ff3828
Oyente	Luu et al. (2016)	github.com/enzymefinance/oyente	480e725
Pakala		github.com/palkeo/pakala	1.1.10
Securify	Tsankov et al. (2018)	github.com/eth-sri/securify	d367b1
teEther	Krupp and Rossow (2018)	github.com/nescio007/teether	04adf5
Vandal	Brent et al. (2018)	github.com/usyd-blockchain/vandal	d2b004

The tools kept in boldface are the ones newly added to the execution framework

4 Analysis Tools

In this section, we specify the selection of tools and describe those that we use in our analysis.

4.1 Selection of Tools

The setting of our study imposes several restrictions on the tools that we can use. Starting from the 140 tools identified by Rameder et al. (2022), we apply the following selection criteria. The numbers in parentheses indicate how many tools in the initial collection fulfill the criterion.

1. Availability: The tool needs to be publicly available with its source open (83).
2. Input: The tool is able to analyze contracts based on their runtime bytecode alone (73). This excludes tools that need access to the application binary interface[7], to the source code, or to a particular state of the blockchain.
3. Findings: The tool offers an automated mode (41) to report weaknesses (79). Some systems are tool boxes for the manual analysis of single contracts. Moreover, not all tools target weaknesses, but contracts like honeypots or Ponzi schemes.
4. Interface: The tool can be controlled via a command-line interface.
5. Documentation: There is sufficient documentation to operate the tool.

Criteria without a number were only checked after the ones with a number had been applied. The selection processes yielded the 12 tools in Table 3.

4.2 Synopsis of Tools

Conkas uses the third-party component Rattle to construct a control flow graph and to lift the bytecode to static single assignments. Then it executes this intermediate representation symbolically and checks the execution traces for patterns that indicate weaknesses.

[7] https://docs.soliditylang.org/en/latest/abi-spec.html

Ethainter uses taint analysis to detect whether attacker-injected data reaches critical operations. It relies on the component Gigahorse that abstracts the bytecode to Datalog rules. The Datalog program, augmented by rules for the addressed weaknesses, is translated to a C++ program (using the tool Soufflé), which is compiled to machine code that performs the actual analysis.

eThor attempts to either prove that the contract is not reentrant and thus is not susceptible to a reentrancy attack, or to find a trace with a reentrant call that indicates that the contract may be vulnerable. eThor lifts the bytecode to constrained Horn clauses that constitute a sound abstraction of the EVM semantics. The Horn clauses are then translated into the constraint language of the SMT solver Z3 that does the final analysis. Various optimizations like the unfolding of Horn clauses are crucial to make the analysis feasible.

MadMax concentrates on gas-related vulnerabilities.[8] Like Ethainter, it uses Gigahorse and Soufflé to lift the bytecode to Datalog.

Maian executes the EVM bytecode symbolically, relying on the SMT solver Z3 to check the satisfiability of path conditions. When analyzing source or deployment code, Maian additionally validates the detected weaknesses by deploying the contract on a private blockchain and attacking it with the transactions computed by the SMT solver. For runtime code this validation step has to be omitted, as there is not enough information for deployment.

Mythril uses symbolic execution, SMT solving (Z3) and taint analysis to detect a variety of security vulnerabilities. To increase code coverage, it applies concolic execution that alternates between symbolic execution and runs with concrete values. It is the only tool in the selection that is actively maintained.

Oyente constructs a control-flow graph and then executes the contract symbolically. The execution traces are checked for patterns characteristic of certain weaknesses. In a final validation step, certain false positives are eliminated before reporting the findings. Symbolic path constraints and the SMT solver Z3 are used to prune the search space.

Osiris extends Oyente by adding modules for further weaknesses.

Pakala executes the bytecode symbolically, without constructing a control flow graph first. It proceeds in two phases. First, it collects transactions that lead to state changes or Ether transfers. Then, the transactions are combined in varying sequences to find one that extracts more Ether than was invested with the transaction.

Securify constructs a control flow graph and lifts the bytecode to single static assignments. It uses Soufflé to derive semantic facts from inference rules specified in Datalog that describe the data and control flow dependencies. These facts are then checked against a set of compliance and violation patterns written in a logic-based domain-specific language.

teEther constructs a control flow graph and searches for paths that lead to critical instructions, with arguments controllable by an attacker. Symbolic execution translates these paths into constraints for the SMT solver Z3. The solutions computed by Z3 can then be turned into exploits, i.e., into transactions that trigger the suspected vulnerabilities.

Vandal constructs a control flow graph and lifts the bytecode to single static assignments. This intermediate representation is translated to logic relations that represent the semantics of the initial bytecode. Weaknesses are specified as Datalog rules. Soufflé synthesizes executable programs that read the logic relations and perform the security analysis.

[8] EVM instructions consume gas proportional to the time and storage they need. Each transaction is endowed with a limit on the total gas it may use, ensuring that each contract terminates.

Table 4 Maintenance Aspects of Tools (checked in February 2023)

Tool	Year	Github issues			Commits		Based on
		open	closed	👤	number	last	
Conkas	2021	8	2	1	3	05.2022	Rattle
Ethainter	2020						Gigahorse, Soufflé
eThor	2020						HoRSt, Soufflé
MadMax	2020	2	5	7	910	06.2021	Gigahorse, Soufflé
Maian	2018	27	8	1	15	03.2018	
Mythril	2020	79	709	74	4785	02.2023	
Osiris	2018	4	2	1	5	09.2018	Oyente
Oyente	2017	63	126	23	848	11.2020	
Pakala	2018	0	12	1	197	03.2020	
Securify	2019	18	53	6	155	09.2019	Soufflé
teEther	2018	2	18	1	21	07.2021	
Vandal	2018	27	10	10	870	07.2020	Soufflé

4.3 Maintenance Aspects

Table 4 lists statistics related to the effort put into keeping the tools up to date. Some tools show a small number of commits and closed issues only, and thus seem unmaintained. But even tools with several hundred commits became unmaintained at some point. Judged by the last code commits, Mythril is the only tool actively maintained: it has 74 contributors, 4 785 commits, and 709 issues resolved. For our study, maintenance mainly boils down to the question which EVM operations the tools are actually able to handle.

4.4 Supported EVM Operations

We say that a tool supports an EVM operation if it models at least the effect of the operation on the stack, by removing and adding an appropriate number of elements. This way the analysis of the current execution path can proceed, even if little may be known about the state after the operation. EVM operations may be unsupported either because of having been omitted deliberately or because of having been introduced by a fork after the tool was released (see Section 3.3). Tools handle unsupported operations by either stopping the analysis of the current path with an error message, by reverting the computation like the EVM would do for an unused opcode, or by aborting with an exception.

The effect of unsupported operations depends on the type of property checked for. Most properties are existential: A bytecode satisfies the property if *some* execution path satisfies a characteristic condition. In such cases an unhandled operation will reduce the number of paths that can be checked. The tool remains sound, but its detection rate diminishes with the number of paths it cannot check. A few properties are universal in nature: The property holds for a bytcode if a condition is met by *all* execution paths. Here, any path that cannot be checked is a threat to the validity of the result, making the method unsound. *Ether lock* is an example for both property types: The weakness is present if there is at least one execution path that increases the balance of the contract (the contract accepts Ether), whereas all execution paths have the property that they do not decrease the balance (the contract offers no withdraw functionality).

To interpret the analysis results later on, we determine the operations supported by each tool. The data in Table 5 was obtained by inspecting the source code as well as by executing the tools on a collection of crafted contracts where each consists of one of the listed operations, with some `PUSH` operations preparing the stack. A checkmark in the table indicates that the tool supports the operation, while a question mark means that the tool supports the operation according to its source code, but fails on the corresponding crafted contract.

The tools usually support the operations up to the most recent fork before their last update. Most tools analyze single contracts only, therefore they handle interactions with other contracts by not supporting the operations `DELEGATECALL`, `STATICCALL`, `CREATE2` and `EXTCODEHASH` at all, or if they do, by invalidating those parts of the state that might have been affected by the operation. Conkas and eThor stick out as handling several operations in their source code that lead to exceptions during execution. Maian is older than it looks: The tool was published some time after the fork at block 4.37 M, yet it does not support the two `RETURNDATA` operations.

Table 5 EVM operations supported by the tools

since block	1.15 M	4.37 M				7.28 M					9.069 M		12.965 M
	DELEGATECALL	RETURNDATACOPY	RETURNDATASIZE	REVERT	STATICCALL	CREATE2	EXTCODEHASH	SAR	SHL	SHR	CHAINID	SELFBALANCE	BASEFEE
Conkas	✓	✓	?	✓	✓	?	?	?	?	?	?	?	
Ethainter	✓	✓	✓	✓	✓	✓		✓	✓	✓			
eThor		✓	✓	✓	?	?	?	?	✓	?			
MadMax	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Maian	✓			✓									
Mythril	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Osiris	✓	✓	✓	✓									
Oyente	✓	✓	✓	✓	✓								
Pakala	✓	✓	✓	✓				✓	✓	✓	✓	✓	
Securify	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			
teEther	✓	✓	✓	✓	✓			✓	✓	✓			
Vandal	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			

A question mark indicates that the operation seems to be handled by the code, but causes an exception

5 Execution Framework

For the large-scale execution of our study, we had the choice between two frameworks: SmartBugs (Ferreira et al., 2020) and USCV (Ji et al., 2021), both operating on Solidity level. We decided on the former, as SmartBugs is better maintained (c.f. the last paragraph of Section 12) and contained more of the tools we were interested in.

First, we adapted SmartBugs to accept bytecode as input, and updated the Docker images of the tools accordingly. Second, we integrated six further tools (kept in boldface in Table 3). The most laborious part was the *output parsers*. For each tool, a dedicated parser scans the output of the tool to identify the result of the analysis, to detect anomalies, and to discard irrelevant messages. For each run of a tool on a bytecode, the parser reports a list of *findings* (tags identifying the detected properties), a list of *errors* (conditions checked for and reported by the tool), a list of *fails* (low-level exceptions not adequately handled by the tool), and a list of *messages* (any other noteworthy information issued by the tool).

Choice of Parameters. Ren et al. (2021) show that the choice of parameters strongly affects the results, especially when the timeout is below 30 minutes per contract. We set the maximal runtime to 1800 s wall time, with 1.5 CPUs assigned to each run. If a tool offers a timeout parameter, we communicate the runtime minus a grace period to allow the tool to terminate properly. Conkas, eThor, Maian, Securify, teEther and Vandal offer no such parameter and are stopped by the external timer.

As there is a tradeoff between the memory limit per process and the number of processes run in parallel, we aimed at providing sufficient but not excessive memory. Based on an initial test with 500 randomly selected contracts, we set the memory limit to 20 GB for

Table 6 Resource Consumption and Out-of-memory (OOM) conditions

Tool	Language	Runtime	Memory	OOM	Re-runs with 32 GB Runtime	OOM
Conkas	Python	338 d	4 GB	319	5 d	44
Ethainter	Python, Datalog	205 d	4 GB	0		
eThor	Java	1651 d	20 GB	534	7 d	369
MadMax	Python, Datalog	61 d	4 GB	0		
Maian	Python	138 d	4 GB	91	1 d	28
Mythril	Python	1926 d	4 GB	50	1 d	18
Osiris	Python	475 d	4 GB	371	6 d	14
Oyente	Python	96 d	4 GB	2229	5 d	1929
Pakala	Python	3204 d	20 GB	421	6 d	333
Securify	Java, Datalog	460 d	20 GB	0		
teEther	Python	1678 d	20 GB	47794	559 d	40306
Vandal	Python, Datalog	172 d	4 GB	2051	23 d	1142

eThor, Pakala, Securify and teEther, and to 4 GB for all other tools. We reran tasks with a limit of 32 GB if they had failed with a segmentation fault or a memory problem.

Machine. We used a server with an AMD EPYC7742 64-Core CPU and 512 GB of RAM. Table 6 gives an overview of the computation time, memory usage, and memory fails before and after the rerun with 32 GB.

6 Weaknesses

In this section, we describe the weaknesses considered in our study as well as the taxonomy used and the mapping of the tool findings to the taxonomy.

6.1 Vulnerability Detection vs. Weakness Warning

According to the Common Weakness Enumeration, cwe.mitre.org, weaknesses are *flaws, faults, bugs, or other errors in software or hardware implementation, code, design, or architecture that if left unaddressed could result in systems, networks, or hardware being vulnerable to attack.*

The tools in our study report findings with varying degrees of certainty, from warnings about potential weaknesses to exploits demonstrating the existence of a vulnerability or, more rarely, proofs guaranteeing their absence. As proving the absence or presence of software properties is difficult, most tools employ heuristics, usually favoring a higher number of false positives over the possibility to overlook an actual vulnerability. Such tools issue warnings and leave the final assessment to the user.

6.2 Synopsis of Weaknesses

Integer Overflow and Underflow (SWC 101) Integer over- and underflow weaknesses arise in situations, where the result of an arithmetic operation exceeds the admissible range and the

system performs a silent wrap-around instead of throwing an exception. The wrong result may lead to an unexpected behavior or a security breach.

Unchecked Call Return Value (SWC 104) This weakness arises when smart contracts do not properly validate the return value of an external contract call, such that an unusual behavior of the call goes unnoticed.

Unprotected Ether Withdrawal (SWC 105) Unprotected Ether withdrawal weaknesses occur when smart contracts allow unauthorized parties to withdraw Ether without proper access controls, risking financial loss.

Unprotected SELFDESTRUCT Instruction (SWC 106) This weakness arises when a `SELFDESTRUCT` instruction is not properly guarded such that an attacker can trigger the destruction of the contract, potentially resulting in the loss of funds or a disruption of services.

Reentrancy (SWC 107) A reentrancy weakness occurs when a contract calls another one without updating its internal state beforehand. If the callee calls the caller back, the latter may be in an inconsistent state.

Assert Violation (SWC 110) Assertions are sanity checks that are meant to hold for every run. An assert violation means that such a check can be made to fail for certain inputs, which can lead to unexpected termination and loss of funds.

Delegatecall to Untrusted Callee (SWC 112) Code invoked by a `DELEGATECALL` instruction operates on the caller's storage and funds. This weakness means that an attacker is able to manipulate the caller's state by controlling the code that is invoked.

DoS with Failed Call (SWC 113) A failing call to an external contract may prevent subsequent actions from taking place. A buggy or malicious callee may cause a DoS with the caller.

Transaction Order Dependence (SWC 114) This weakness arises when a contract's behavior depends on the order in which transactions are mined, leading to inconsistencies and potential security issues.

Authorization through tx.origin (SWC 115) This weakness means that a contract relies on tx.origin for authorization, which can be exploited by a man-in-the-middle attack to bypass access controls.

Block Values as a Proxy for Time (SWC 116) This weakness results from using block-related values as a substitute for precise timing, as these values can be manipulated by attackers.

Weak Sources of Randomness from Chain Attributes (SWC 120) Seeding random number generators with chain attributes leads to weak randomness, as these attributes can be predicted or even manipulated by attackers.

Write to Arbitrary Storage Location (SWC 124) This weakness occurs when an attacker is able to write to an unintended storage location, e.g. by overflowing one data structure into the next in storage.

Arbitrary Jump with Function Type Variable (SWC 127) Solidity supports function types. By low level manipulations of variables that hold a function, control can be handed over to code other than the function, leading to unintended execution paths.

DoS With Block Gas Limit (SWC 128) The resource consumption of every call to a contract is limited by the gas supplied. But this gas limit is capped itself by the block gas limit. Once the call to a contract requires more gas than that, e.g. because of looping over a data structure that has been grown too big by an attacker, the contract may become inoperable.

Ether Lock This weakness means that a contract accepts Ether without offering the functionality to withdraw it, thus locking any funds sent to it.

Callstack Depth Bug Originally, the number of nested calls was limited to 1024. Calling a contract at this limit would make any further nested calls fail unexpectedly, leading to a

Table 7 Weakness Classes and Reports

Weaknesses	SWC	Frequency	Tools
Integer Overflow and Underflow	101	123255	4
Unchecked Call Return Value	104	183277	4
Unprotected Ether Withdrawal	105	17916	7
Unprotected SELFDESTRUCT Instruction	106	7310	4
Reentrancy	107	184610	7
Assert Violation	110	62246	1
Delegatecall to Untrusted Callee	112	1352	3
DoS with Failed Call	113	20202	2
Transaction Order Dependence	114	35875	4
Authorization through tx.origin	115	8669	2
Block Values as a Proxy for Time	116	46372	4
Weak Sources of Randomness from Chain Attributes	120	8733	1
Write to Arbitrary Storage Location	124	8826	2
Arbitrary Jump with Function Type Variable	127	2474	1
DoS With Block Gas Limit	128	8863	1

potentially harmful situation for the called contract. As early as block 2.463 M, this limit was replaced by a better mechanism that made the weakness related to the callstack depth limit obsolete.

6.3 Mapping of Tool Findings

Taxonomy. To compare the tools regarding their ability to detect weaknesses, we need a taxonomy with an adequate granularity. Since there is no established taxonomy of weaknesses for smart contracts, previous studies (Chen et al., 2020; Tang et al., 2021; Wang et al., 2021; Kushwaha et al., 2022; Rameder et al., 2022; Tolmach et al., 2022; Zhou et al., 2022) not only summarize potential issues, but also structure them with respect to their own taxonomies, none of which is compelling or widely used.

Among the community projects, there are two popular taxonomies: the DASP (Decentralized Application Security Project) TOP 10 [9] from 2018, which features 10 categories, and the SWC registry (Smart Contract Weakness Classification and Test Cases) [10] with 37 classes, last updated 2020. As for DASP, two categories, *Access Control (2)* and *Other (10)*, are quite broad, while *Short Address (9)* is checked by hardly any tool. Moreover, *DOS (5)* and *Bad Randomness (6)* are effects that may be the result of various causes, and most tools detect causes rather than consequences.

The SWC registry is more granular as it offers several classes for the broad categories *Access Control* and *DOS*. Moreover, most of its categories match relevant findings of the tools. Therefore, we select this taxonomy as the basis of our comparison.

Findings mapped. The tools report 82 different findings, of which we can map 56 to one of the 37 classes of the SWC taxonomy (see Table 15 in the appendix). In total, the tools cover 15 SWC classes. Table 7 lists the weakness classes, the accumulated number of findings that

[9] https://dasp.co

[10] https://swcregistry.io

Table 8 SWC Classes Detected by Tools

Tool	101	104	105	106	107	110	112	113	114	115	116	120	124	127	128	total
Conkas	✓	✓			✓				✓		✓					5
Ethainter			✓	✓			✓						✓			4
eThor					✓											1
MadMax	✓							✓							✓	3
Maian			✓	✓												2
Mythril	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		13
Osiris	✓				✓				✓		✓					4
Oyente					✓				✓		✓					3
Pakala			✓				✓									2
Securify		✓	✓		✓				✓							4
teEther			✓													1
Vandal		✓	✓	✓	✓					✓						5

the tools report, and the number of tools that address the weakness. Table 8 gives an overview of the coverage of the SWC classes by tool.

When a tool reports a finding, we assume that it is not invalidated by an accompanying error condition, a low coverage of the bytecode, or a timeout. However, we note errors, timeouts, and unhandled conditions (fails).

Findings omitted. In order not to count the same weakness twice, we exclude redundant and intermediate findings. Moreover, some findings state the absence of a weakness and thus should not be counted as a weakness. Altogether, we exclude seven findings: For eThor the positive finding *secure* (from reentrancy); for Maian the intermediate finding *accepts_Ether* and the positive findings *no_Ether_leak*, *no_Ether_lock*, *not_destructible*; for Osiris the redundant finding *arithmetic_bug* (as it is doubled by a more specific one); and for Vandal the preliminary finding *checked_call_state_update*.

7 RQ1 Abstraction

To validate our hypothesis that bytecodes with the same skeleton (i.e., members of the same code family) behave similarly regarding bytecode analysis, we randomly select 1 000 bytecodes from all runtime bytecodes *not* in our data set. By construction, these codes belong to families with at least two members. The selected bytecodes happen to belong to 620 families. We add the corresponding 620 representatives from our data set, obtaining a dataset with 1 620 bytecodes and 620 families with 2 to 64 members per family.

When running analysis tools on different members of the same family, we expect nearly identical results with small variations due to differences in runtimes (e.g. one run timing out while the other one finishes just in time with some finding) or due to the effect of different constants when solving constraints. In particular, we do not expect the meta-data injected by the Solidity compiler to affect the result, as it is interpreted neither as code nor as data during execution. To confirm this, we also consider a copy of our 1 620 bytecodes, where we replace all meta-data sections with zeros.

Table 9 shows the result of running all tools on the bytecodes with and without metadata. Columns two and four give the percentage of the 620 families for which the findings

differ within the family, whereas columns three and five consider all data collected by the output parsers, including errors, fails, and messages. If we assume that the various effects influencing the output give rise to a normal distribution, then for a confidence level of 95%, the sample size of 620 yields a margin of error of 1.5% for the smaller values in the table and of 3.2 % for the larger ones.

The seven tools on top behave essentially as predicted. For Conkas, the rate of 1.5 % corresponds to 9 families with divergent findings. These differences are related to warnings about integer under- and overflows, and may indeed be the result of different constants in the codes of a family. Observe that for these seven tools, there is hardly any difference between the two datasets, with and without meta-data.

Osiris and Oyente seem remarkable, as we find 20% discrepancies in the output. Oyente starts its analysis by disassembling the entire bytecode. It issues the warning 'incomplete push instruction' when stumbling upon a supposed `PUSH` instruction near the end of the meta-data that is followed by too few operand bytes. These spurious messages disappear when removing the meta-data, but otherwise do not affect the analysis. Osiris reuses Oyente's code and inherits this anomaly.

eThor also scans the entire bytecode. When encountering an unknown instruction, it issues a warning and ignores the remaining code. Like with Oyente, these messages mostly disappear when removing the meta-data. However, unlike Oyente, the meta-data influences the result of the analysis, as can be observed by 2.9 % vs. 1.0 % differences in the findings for code with vs. no meta-data. In each of these cases, the analysis times out for some member(s) of

Table 9 Code Families with Diverging Results [%]

Tool	with meta-data		without meta-data	
	findings	all fields	findings	all fields
Conkas	1.5	1.9	1.3	1.9
Ethainter	0.0	0.2	0.0	0.3
MadMax	0.0	0.0	0.0	0.5
Mythril	0.2	0.6	1.0	1.6
Pakala	0.0	1.8	0.0	1.8
Securify	0.0	0.0	0.2	0.5
teEther	0.0	2.3	0.0	2.3
Oyente	0.2	20.0	0.3	0.5
Osiris	1.6	21.3	0.8	1.3
eThor	2.9	8.2	1.0	1.5
Vandal	3.2	4.7	0.2	2.1
Maian	11.3	11.5	0.5	0.6

the family but terminates with identical results for the others. We did not research the cause for these discrepancies but suspect that it may be comparable to the situation of Vandal.

Vandal constructs a control flow graph for the entire bytecode and decompiles it to an intermediate representation. Sometimes, the tool gets lost during this initial phase and times out. The situation improves when removing irrelevant parts like the meta-data. However, as Vandal interprets the addresses of all code sections relative to the beginning of the bytecode, even if they belong to a different contract (see the discussion on the structure of bytecode in Section 3.1), we still see differences regarding errors and fails.

Maian starts by scanning the entire bytecode for certain instructions, like `SELFDESTRUCT`. Not detecting the opcode anywhere lets Maian immediately conclude certain properties, whereas finding the opcode triggers a reachability analysis that may remain inconclusive. This sensitivity to single bytes yields divergent results for 70 families. For example, Maian may detect non-destructibility for one code and fail to do so for another one in the same family. Removing the meta-data gets rid of these divergences almost entirely.

Observation 1. Treating bytecodes with the same skeleton as equivalent works for 9 out of 12 tools without reservations. Three tools unexpectedly analyze the meta-data, leading to minor output variations. Therefore, skeletons can be regarded as a suitable abstraction for large-scale analyses aimed at the big picture. Removing the meta-data prior to analysis may improve the performance of some tools (while not harming others).

8 RQ2 Weakness Detection over Time

In this section, we portray the evolution of weaknesses on a timeline of blocks. We look at the percentage of contracts flagged by a particular tool as possessing any weakness (Fig. 2) as well as at the percentage of contracts flagged by any tool as possessing a particular weakness (Fig. 4).

8.1 Tool reports

Figure 2 depicts the reporting rate of each tool over the range of 14 M blocks. Each data point represents the percentage of bytecodes in a bin of 100 k blocks that were marked with at least one non-omitted finding by the respective tool. The vertical lines in gray indicate forks that added EVM opcodes and thus may affect weakness detection. To improve readability, the diagram is split into three plots with four tools each.

Upper plot. Pakala (green) and teEther (red) both flag a few bytecodes only. This can be attributed to the fact that Pakala scans for two rather infrequent weaknesses (SWC 105, 112), and teEther just for one (SWC 105). eThor (orange) also scans for a single weakness (SWC 107), albeit for a far more prevalent one. Conkas (blue) scans for five weaknesses (SWC 101, 104, 107, 114, 116), among them the most frequent ones (SWC 101 and 107).

Middle plot. MadMax (blue) and Ethainter (orange) specialize in rather specific weaknesses that they detect for a small number of contracts only. MadMax is geared towards three gas issues, loosely related to SWC 101, 113, and 128, while Ethainter scans for five weaknesses (SWC 105, 106, 112, 124, and unchecked tainted static call). Securify (red) also scans for five weaknesses (SWC 104, 105, 107, 114, and missing input validation), including the popular reentrancy bug.

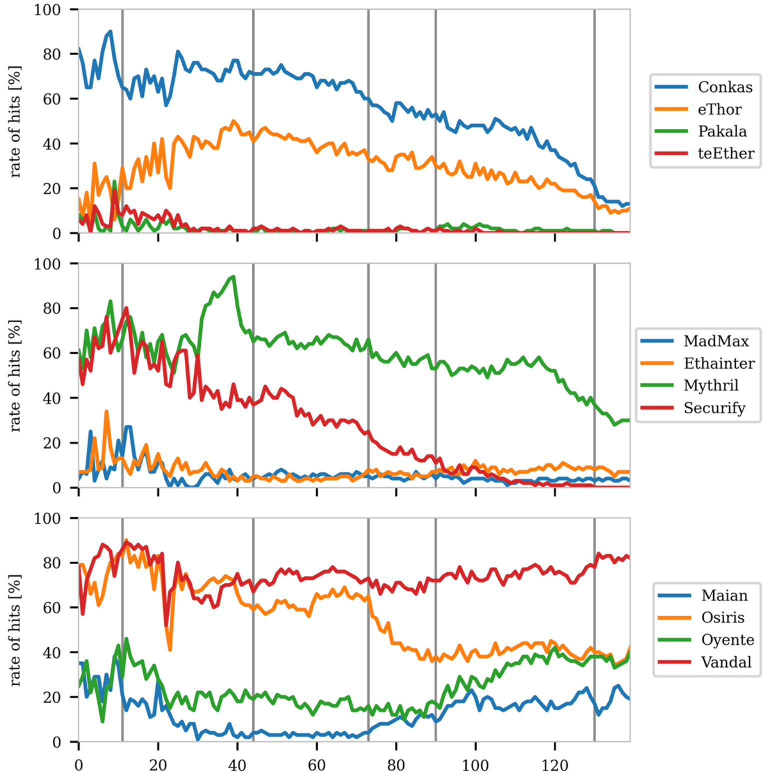

Fig. 2 Accumulated findings per tool over time. Each data point shows the percentage of bytecodes for which the tool reports a weakness, in bins of 100 k blocks

Mythril (green) tests for the largest number of weaknesses (SWC 101, 104–107, 110, 112, 113, 115, 116, 120, 124, 127), including the most prevalent ones. While we see a peak with more than 90% of contracts flagged in the early days, the rate of contracts with reported weaknesses continuously drops to below 40% towards the end of the timeline.

Lower plot. The tools in this plot differ from the others, as the rate of flagged contracts stays high or even increases towards the end of the timeline. Maian (blue) scans for three weaknesses (SWC 105, 106, and locked Ether), Osiris (orange) for nine (SWC 101, 107, 114, 116, integer issues beyond SWC 101, and the callstack depth bug), Oyente (green) for four (SWC 107, 114, 166, and the callstack depth bug), and Vandal (red) for five (SWC 104–107 and 115).

In terms of EVM operations supported (see Section 4.4), Maian, Oyente and Osiris are the oldest tools in our collection. They do not handle the operation `SHR`, which is central to newer contracts (Fig. 1). Hence, we expect the rate of findings to drop over time rather than to rise. It turns out that Oyente checks for *Callstack Depth Bugs* by searching for a specific code pattern (instead of using symbolic execution as for the other weaknesses), and Osiris inherits this functionality from Oyente. Even though the bug has become obsolete with the fork at block 2.463 M (see Section 6.2), the pattern is detected at an increasing rate and causes

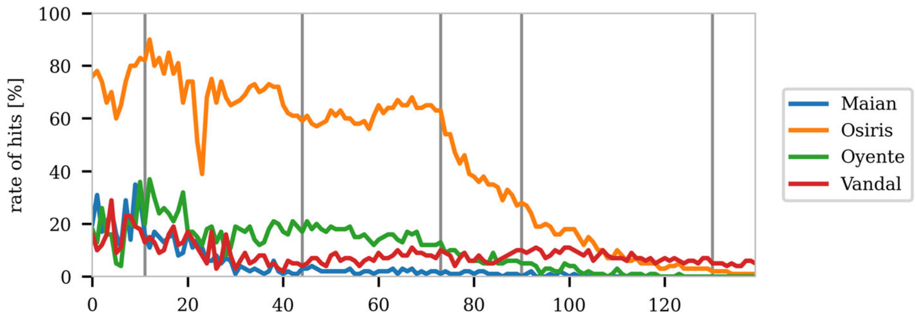

Fig. 3 Accumulated findings for Maian, Osiris, Oyente and Vandal. Compared to the third plot in Fig. 2, some spurious findings have been omitted (see the text for details)

spurious findings. Regarding Maian, it checks, among other weaknesses, for *Ether lock*. This property requires to check all execution paths for the absence of operations that are able to transfer Ether. As the inability to handle SHR cuts short more and more of the paths, the number of falsely reported Ether locks increases.

Vandal reports 96.6 % of the contracts with a CALL instruction as containing an *Unchecked Call* and 88.4 % as containing a *Reentrant Call*. Given that the majority of calls are method calls, for which the Solidity compiler adds checks automatically, and given that reentrancy is known to be a common but not a universal problem, we suspect that Vandal applies weak criteria and thus reports numerous false positives.

In Fig. 3, we omit the problematic findings *Callstack Depth Bug* for Oyente and Osiris, the finding *Ether lock* for Maian, and the weaknesses *unchecked call* (SWC 104) and *reentrant call* (SWC 107) for Vandal. With these omissions, the number of flagged contracts either is constantly low or drops low.

General Observation. Overall, the share of flagged contracts diminishes over time. For unmaintained tools, this may be related to the fact that they are no longer able to analyze recent contracts containing e.g. new instructions. Moreover, code patterns tailored to specific compiler versions may fail to detect a weakness in bytecode obtained by later versions. For actively maintained or new tools, the decreasing number of flagged contracts may indeed indicate that newer contracts are less vulnerable than older ones.

8.2 SWC classes detected

For weaknesses mapped to a suitable SWC class, Table 7 gives an overview of their prevalence. The column *frequency* counts the number of unique skeleton bytecodes, where at least one tool reports the respective weakness[11]. As the tools tackle differing subsets of the SWC classes, the number of tools addressing a specific weakness varies from one to seven. Due to our cumulative counting, the frequency of a weakness increases with the number of tools claiming to detect it, especially with overreporting tools.

Figure 4 depicts the 15 SWC classes on the timeline of 14 M blocks. For every SWC class, a data point represents the percentage of skeleton bytecodes in a bin of 100 k blocks

[11] Most tools do not verify their assessment by providing an exploit (like teEther does) or by proving the absence of the vulnerability (like eThor does). Hence, the table counts warnings rather than vulnerabilities.

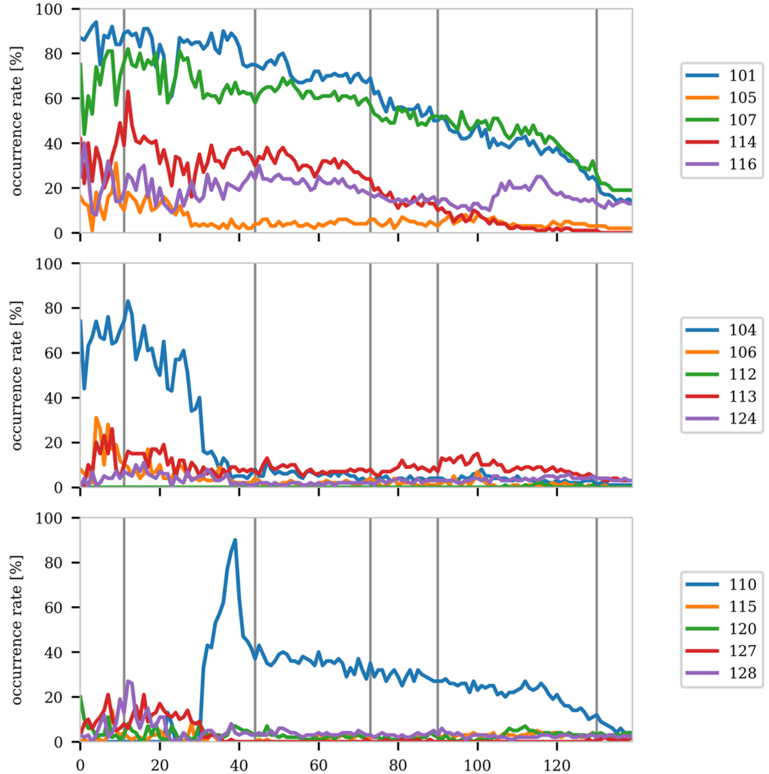

Fig. 4 SWC classes over time. Each data point shows the percentage of bytecodes flagged with a specific weakness, in bins of 100 k blocks

that were marked with the respective weakness by at least one tool. The top plot shows the classes detected by four or more tools (SWC 101, 105, 107, 114, 116), the middle one those handled by two or three tools (SWC 104, 106, 112, 113, 124), and the third one those addressed by just one (SWC 110, 115, 120, 127, 128). In accordance with our discussion of Vandal above, we omit its findings from the plots, as its excessive reporting for SWC 104 and 107 would distort the picture.

We see five weaknesses decrease over time from a high ($\geq 50\,\%$) or medium ($20\,\%$) level to a medium or low ($\leq 10\,\%$) level: The findings of classes 101, 104, 107, 110, and 114 start to fall from about block 4 M onwards. The other 10 weaknesses stay on a steady, but low level after block 4 M, except for 113 (middle plot), which fluctuates around $10\,\%$ and 116 (top plot), which fluctuates around $20\,\%$.

The decline of potential integer overflows (101) seems plausible: Since version 0.8.0, the Solidity compiler adds appropriate checks automatically, and already some time before, the use of math libraries with the same effect had become quasi-standard. Reentrancy (107) is probably the most (in)famous vulnerability. The decrease in detection can be attributed at least partially to developers taking adequate precautions.

Table 10 Average Runtimes of Tools

Tool	Overall	Success	Error	OOM	Prg.issues
Conkas	119 s	84 s	28 s	719 s	21 s
Ethainter	71 s	20 s	1753 s		
eThor	574 s	90 s	17 s	874 s	10 s
MadMax	21 s	21 s	1752 s		
Maian	48 s	58 s	18 s	1104 s	62 s
Mythril	670 s	660 s		834 s	129 s
Osiris	165 s	165 s	165 s	1242 s	290 s
Oyente	35 s	34 s		171 s	30 s
Pakala	1115 s	785 s	4 s	1154 s	627 s
Securify	160 s	122 s	19 s		940 s
teEther	572 s	72 s	754 s	886 s	441 s
Vandal	63 s	41 s		665 s	26 s

Observation 2. Of the 37 SWC classes, 15 are covered by at least one tool, and 7 by at least three tools. For all weaknesses, the number of flagged contracts decreases over time or stagnates on a low level. The decreasing detection rates can be attributed to unmaintained tools that do not adequately cope with newer EVM instructions as well as to compilers and programmers taking counter-measures. At the end of the timeline, *integer bugs* (SWC 101), *reentrancy* (SWC 107) and *block values as a proxy for time* (SWC 116) are the most frequently detected weaknesses with a share of about 20% each.

9 RQ3 Tool Quality over Time

To assess the quality of the tools, we consider execution times as well as their errors and failures.

Execution time. Table 10 gives the average runtimes in seconds for each tool. The column *Overall* averages over all 248 328 runs, whereas *Success* picks only those completing without errors and failures. The column *Error* shows the average time for runs where the tool reports an error, while *OOM* collects the runs terminated by an out-of-memory exception. The last column, *Prg.issues*, averages over runs with programming issues, like exceptions caused by type errors. The average time for runs timing out is not listed explicitly, as it is close to 1 800 s (30 m), for obvious reasons.

Overall, the fastest tools are MadMax, Oyente, Maian, Vandal, Ethainter. The slowest one, by far, is Pakala, with the next ones, Mythril, eThor, and teEther, being twice as fast. When considering only runs without errors and failures, eThor and teEther are substantially faster than on average, while Pakala and Mythril are still slow. Mythril, Oyente, and Vandal do not report any errors, hence no times are listed in the respective column. The average times on error are small for Pakala, Securify, Maian, eThor and Conkas, which indicates that most

reported errors are show-stoppers. For Madmax and Ethainter, the few errors are related to a timeout, hence the average is high.

Errors and Failures. We consider a run failed if it is terminated by an external timeout, an out-of-memory exception, or a tool-specific unhandled condition. A run terminates properly if it stops under control of the tool, either successfully or with an error condition detected by the tool. Figure 5 depicts the failures over time in bins of 100 k blocks as percentage of bytecodes where a tool fails. Conkas, eThor, Pakala, and teEther fail most often and are shown at the top, while the other tools show few or no failures.

Figure 6 depicts the error over time in bins of 100 k blocks as percentage of bytecodes where a tool reports an error. Maian and Osiris show an increasing error rate, while the other tools show few or no errors. Table 11 gives an overview of the accumulated errors and failures by category. The left part lists the number of bytecodes with and without finding, as well as the share accompanied by an error or failure. In its right part, the table gives the number of bytecodes, where the analysis resulted in an error message and/or a failure due to a timeout, an out-of-memory condition or a program issue.

While most reported findings are not accompanied by any errors or failures, there are three notable exceptions. Maian detects numerous occurrences of *Ether lock* in spite of encountering unknown instructions. The same accounts for Osiris when it reports the *Callstack*

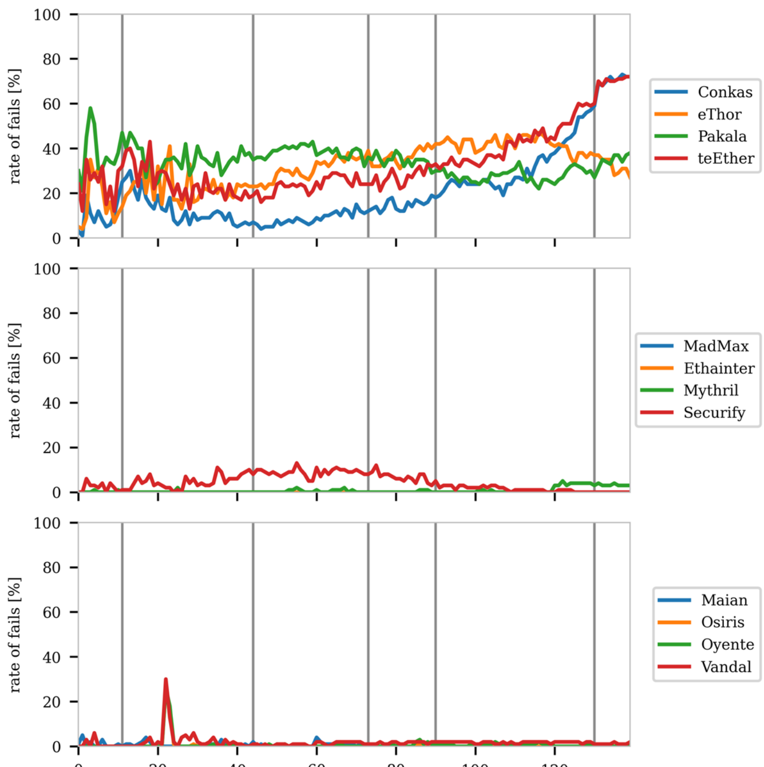

Fig. 5 Tool failures over time. Each data point shows the percentage of failures encountered by the tools, in bins of 100k blocks. Ethainter and MadMax had no failures

bug. This is due to the fact that the tools apply local pattern matching instead of symbolic execution. Pakala reports a timeout for almost half of its analyses with findings.

eThor, Pakala and teEther show a large number of timeouts (marked red in Table 11), which results in high average runtimes (marked red in Table 10). While Mythril shows a similarly high average runtime, it only has a low number of timeouts. In contrast to the other three tools, it offers a parameter for getting notified about the external timeout and so is able to finish in time.

Regarding out-of-memory exceptions, only teEther sticks out. Even with 32 GB of memory, it still fails for 16 % of the inputs.

The last column in Table 11, program issues, indicates to some extent the maturity of the tools' code. Conkas fails for 63 111 runs, with the most common causes being *maximum recursion depth exceeded* (55626 / 88.1%); assertion failures (2499 / 4.0%); and type errors (2038 / 3.2%). The 17 407 fails of eThor result from the instruction EXTCODEHASH not being processed properly (11366 / 65.3%), arithmetic exceptions (5930 / 34.1%), and runtime exceptions (111 / 0.6%). Securify fails for 9 586 runs, mainly because of *null pointer* exceptions (9496 / 99.1%). Mythril, as the only tool actively maintained according to the activity on Github, fails for only 1 022 bytecodes, the most frequent cause being type errors (952 / 93.2%), predominantly due to undefined terms in integer expressions. At the lower end, we find Ethainter and MadMax with no program issues at all, and Maian, Osiris and Oyente with just a few.

Observation 3. Regarding resource consumption, a few tools require less than 60 s per contract with just a few GB of memory, whereas others regularly approach the limits of 30 min and 32 GB. The rate of tool-reported errors varies between 0 % and 60 %, with the high rates resulting from tools operating outside of their specification. Questionably, there are tools with similar limitations but without any error at all. Regarding robustness, eight tools throw an exception for less than 1 % of the contracts, as opposed to one tool with 25 % fails. Program issues like type exceptions may be a consequence of using the dynamically typed language Python.

Table 11 Findings, Errors and Failures of Tools

	Bytecodes with/without findings				Bytecodes with errors/failures			
Tool	with	+err/fail	without	+err/fail	Error	Timeout	OOM	Prg.issues
Conkas	121 436	0 %	126 892	69 %	16 790	7 825	44	63 111
Ethainter	17 842	0 %	230 486	3 %	7 290	0	0	0
eThor	148 643	0 %	99 685	99 %	9 892	71 290	369	17 407
MadMax	11 868	0 %	236 460	0 %	13	0	0	0
Maian	31 929	84 %	216 399	41 %	11 4826	1 298	28	10
Mythril	135 818	0 %	112 510	3 %	0	2 620	18	1 022
Osiris	122 883	53 %	125 445	69 %	15 1010	86	14	576
Oyente	67 752	0 %	180 576	1 %	0	11	1 929	563
Pakala	4 232	46 %	244 096	34 %	10	80 808	333	2 574
Securify	40 699	3 %	207 629	10 %	10 525	1 651	0	9 586
teEther	3 230	3 %	245 098	41 %	3 752	52 250	40 306	6 608
Vandal	187 506	0 %	60 822	8 %	0	2 662	1 142	1 047

10 RQ4 Overlap Analysis

In this section, we investigate to which extent the tools agree in their judgments. We use the SWC registry as a common frame of reference and map all findings to an appropriate SWC class, if any.

This excludes findings that do not fit any SWC class. More specifically, the following nine findings are omitted for that reason: one finding of Ethainter (*unchecked_tainted_static call*), one of Securify (*missing_input_validation*), one of Maian (*Ether_lock*), five of Osiris (*Callstack_bug*, *Division_bugs*, *Modulo_bugs*, *Signedness_bugs*, *Truncation_bugs*), and one of Oyente (*Callstack_Depth_Attack_Vulnerability*).

To determine the degree of overlap, we use the following measure. For a tool t, let $\mathrm{Swc}(t)$ be the set of SWC classes that t is able to detect, and let $\mathrm{Flagged}(t, s)$ be the set of contracts that t flags for having a weakness of class s. We define the overlap between two tools t_1 and t_2 as

$$\mathrm{Overlap}(t_1, t_2) = \frac{\sum_{s \in \mathrm{Swc}(t_1) \cap \mathrm{Swc}(t_2)} |\mathrm{Flagged}(t_1, s) \cap \mathrm{Flagged}(t_2, s)|}{\sum_{s \in \mathrm{Swc}(t_1) \cap \mathrm{Swc}(t_2)} |\mathrm{Flagged}(t_1, s)|}$$

The numerator counts, per weakness, the contracts flagged by both tools, while the denominator gives the number of all contracts flagged by the first tool. This measure is not symmetric. $\mathrm{Overlap}(t_1, t_2) = 100\%$ means that for the SWC classes in common, t_1 flags a subset of the contracts flagged by t_2. If additionally $\mathrm{Overlap}(t_2, t_1) = 100\%$ holds, then the two tools are in perfect agreement, something to be expected for $t_1 = t_2$ only.

Table 12 shows the overlap between any two tools, with t_1 listed vertically and t_2 horizontally. Since eThor detects reentrancy only, its row and column in the table give an idea of how differently a weakness may be assessed by the tools. For Vandal, we find high values in its column and low ones in its row, which indicates that most weaknesses it reports are not backed by other tools. As discussed in Section 8.1, a large number of Vandal's findings are likely to be false positives, and the numbers in Table 12 reflect that.

Another observation concerns Osiris and Oyente. We expect a high overlap as Osiris extends Oyente. In fact, 90.2 % of Oyente's findings are backed by Osiris, while Oyente covers 58.5 % of Osiris' findings. Apparently, Osiris not only detects additional weaknesses (not considered in the comparison), but also flags additional contracts with weaknesses the tools have in common.

Figure 7 shows the overlap in more detail. We exclude Vandal (due to its overreporting) and Oyente (as Osiris extends it), to avoid an inflation of overlaps. Each row gives a breakdown of the contracts flagged by a specific tool, for each SWC class covered by at least two tools. Blue identifies the share of contracts flagged exclusively by the tool, whereas red, green, and purple indicate the share also flagged by one, two, or more other tools. A good agreement shows as purple where four or more tools check for the SWC class (101, 105, 107), green where three tools detect it (104, 106, 112, 114, 116), and red for two tools (113, 124).

SWC 101 – Integer Overflow and Underflow: We find hardly any agreement of all four tools. MadMax, by construction, checks for a subcase of 101 that is not covered by the other tools, but even green (overlap of three) is rare. In Section 11.1, we analyze the evolution of overlaps for this weakness in more detail.

SWC 104 – Unchecked Call Return Value: The three tools show some agreement, as red and green dominate blue.

Table 12 Overlap of Tool Findings [%]

$t_1\downarrow$ $t_2\rightarrow$	Conkas	Ethainter	eThor	MadMax	Maian	Mythril	Osiris	Oyente	Pakala	Securify	teEther	Vandal
Conkas	100.0		56.8	1.6		28.7	26.2	6.7		14.8		85.1
Ethainter		100.0			14.3	15.6			17.1	4.7	13.0	31.4
eThor	68.3		100.0			22.2	5.6	2.3		6.6		76.0
MadMax	39.2			100.0		24.9	46.6					
Maian		38.4			100.0	65.3			57.6	17.9	81.5	44.9
Mythril	52.3	35.9	49.6	1.5	28.9	100.0	22.3	5.8	60.0	9.0	34.9	85.7
Osiris	58.4		49.8	2.8		35.6	100.0	58.5		44.0		94.5
Oyente	35.1		54.7			55.2	90.2	100.0		50.0		93.5
Pakala		24.6			29.4	61.6			100.0	9.0	44.9	23.8
Securify	38.1	13.2	50.6		21.0	17.7	41.0	29.8	20.6	100.0	27.3	70.9
teEther		19.3			50.2	43.2			54.2	14.4	100.0	30.7
Vandal	24.1	21.2	33.4		11.3	11.4	4.7	1.7	11.0	4.0	11.8	100.0

SWC 105 – Unprotected Ether Withdrawal: Detected by six tools, we see the highest amount of purple among all classes.

SWC 106 – Unprotected SELFDESTRUCT Instruction: Virtually all of Maian's findings coincide with at least one other tool, while Ethainter and Mythril show a fair amount of blue.

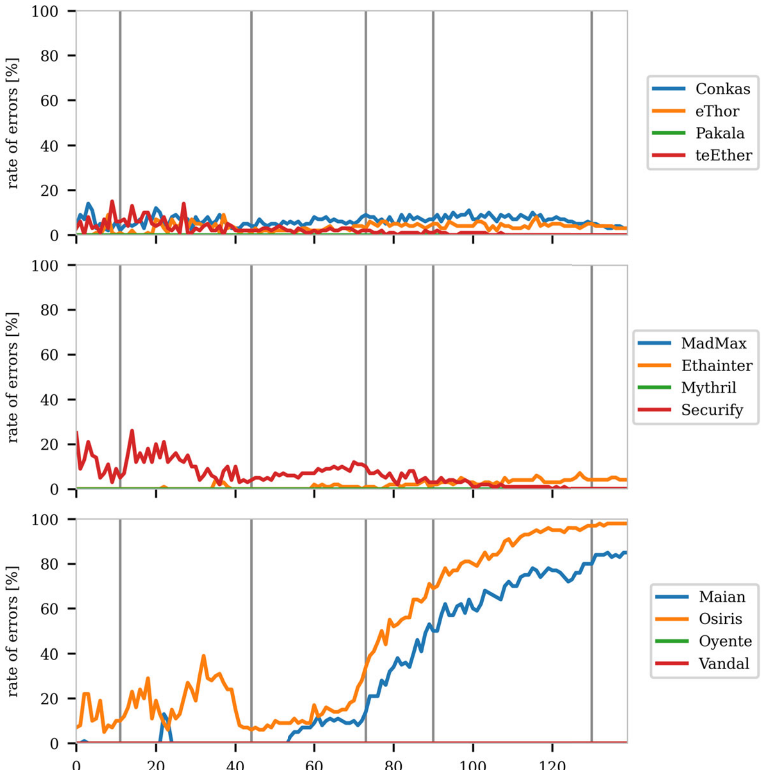

Fig. 6 Tool errors over time. Each data point shows the percentage of errors reported by the tools, in bins of 100k blocks. Mythril, Oyente and Vandal had no errors

Fig. 7 Agreement of the tools' judgment on the SWC classes. Each bar shows the proportion of weaknesses identified by one ■, two ■, three ■, and four or more ■ tools

The top plot of Fig. 6 provides an explanation: In the second half of the timeline, the error rate of Maian increases, as the tool fails to handle more recent contracts with new types of instructions, so Maian stops reporting weaknesses.

SWC 107 – Reentrancy: Even though reentrancy is one of the best-researched weaknesses and is detected by five tools, agreement of more than three tools is rare. In Section 11.1, we analyze the evolution of overlaps for this weakness in more detail.

SWC 112 – Delegatecall to Untrusted Callee: This weakness is detected by three tools, hence the large amount of green actually indicates the best agreement in the chart. Ethainter seems to implement a more liberal definition of the vulnerability, as it flags many additional contracts (blue).

SWC 113 – DoS with Failed Call: MadMax has been designed to detect specific gas-related issues, which partly map to this class. There is some overlap with Mythril, but since the latter flags many more contracts under this label, the red share is not visible in Mythril's bar.

SWC 114 – Transaction Order Dependence: The bars are mainly blue and red, indicating little agreement between all three tools.

SWC 116 – Block Values as a Proxy for Time: Virtually all contracts flagged by Osiris are also flagged by one of the other tools, in most cases by both. The other tools, however, flag many more contracts, as the comparatively small size of the green part – representing the same group of contracts in all three bars – shows. Like in the case of Maian and SWC 106 above, the error rate of Osiris increases in the second half of the study period, as new instructions prevent it from reporting weaknesses (Fig. 6).

SWC 124 – Write to Arbitrary Storage Location: The contracts flagged by Mythril are essentially a subset of those flagged by Ethainter, but a small one, as the blue part of Ethainter's bar dominates.

Observation 4. There is little agreement between the tools regarding the findings, even for well-researched and frequently analyzed weaknesses such as reentrancy. Contributing factors are the lack of commonly accepted, precise definitions for the weaknesses as well as diverging approaches to detect them. A mutually low agreement suggests that the tools are rather complementary.

11 Discussion

In this section, we combine the results of our research questions and discuss them in a wider context.

11.1 Relation between Findings, Errors, Failures, and Overlap

In the last section, we looked at the overlap of tools, accumulated over time as well as over common SWC classes. Here, we pick two exemplary SWC classes, take a closer look at the evolution of findings over time, and correlate the overlap of tools with their errors and failures.

SWC 101 – Integer Over- and Underflow. In Fig. 8, the top most plot depicts the percentage of bytecodes flagged with an integer over- or underflow, per tool. Starting from different levels around 70 % and 40 %, Conkas and Mythril converge at 10 % at the end of the timeline. Osiris shows a weakness level comparable to these tools for most of the timeline, but falls to 0 % towards the end. MadMax reports hardly any cases throughout the whole timeline.

Fig. 8 SWC-101 Integer Overflow and Underflow on a timeline of blocks, in bins of 100 k blocks. Top: Percentage of bytecodes flagged, per tool. Upper middle: Percentage of overlaps. Lower middle: Error rate of tools. Bottom: Failure rate of tools

The second plot in Fig. 8 visualizes the agreement of tools over time. The lines in the foreground show the number of contracts flagged by any tool, once in relative terms (blue line with the scale to the left, with 100% corresponding to all contracts), and once in absolute terms (black line with the scale to the right, numbers per bin of 100 k blocks). The background divides the flagged contracts into shares that are flagged by a single tool, by two, three, or four tools, respectively.

Up to block 6 M (bin 60), the brown and green areas with purple specks at the top show that 60% of the flagged contracts are flagged by at least two tools. The other 40% are split between Osiris and Conkas, who are the sole tools flagging the respective contracts. The gray area of contracts flagged solely by Mythril is small, even though the tool finds the weakness in 20–40% of all contracts (top plot). Apparently, at least one other tool agrees with Mythril most of the time.

The picture changes in the second half of the plot. Towards the end of the timeline, there is hardly any agreement anymore. Less than 10% of the contracts are flagged by at least two tools, while most are flagged solely by Conkas or Mythril.

The situation can be partly explained by the fact that MadMax specializes in gas issues, with one of its findings constituting a specific type of overflow that occurs in a few contracts only (see top plot). For Osiris, we see a rise in errors (third plot of Fig. 8) that mirrors the increased usage of the `SHR` operation (Fig. 1), which is not supported by Osiris (Table 5). Therefore, the detection rate of Osiris drops to zero (top plot), leaving us essentially with two tools at the end of the timeline. In spite of Conkas' failure rate rising to 70% (fourth plot of Fig. 8), its detection rate remains comparable to Mythril.

From version 0.8.0 onwards, the Solidity compiler inserts checks for over- and underflows into the bytecode. In view of the compiler's adoption rate (Fig. 1), it seems that the vulnerability has actually become extinct at the end of the timeline and that the respective findings of Mythril and Conkas are false positives.

SWC 107 – Reentrancy. At a first glance, Fig. 9 shows a situation similar to Fig. 8, just for another weakness and the six tools detecting it. The detection rate of three tools (Osiris, Oyente, Securify) is low and drops to zero towards the end. For Osiris and Oyente, the reason is again their inability to handle new operations, in particular `SHR`, even though Oyente quits silently, while Osiris issues errors (third plot). For Securify, the collected data does not provide an explanation for the diminishing detection rate.

Conkas and eThor exhibit significant failure rates (bottom plot in Fig. 9), but this does not prevent them from reporting up to 40% of contracts as potentially vulnerable to a reentrancy attack. On the surface, these two tools show a similar behavior, reporting similar rates of reentrant contracts from block 3.5 M (top plot, bin 35) onwards. However, the Jaquard similarity for the flagged contracts (number of contracts flagged by both tools divided by the number of contracts flagged by at least one tool) is only 45 % at block 3.5 M, and drops to 28 % for the last part where the blue and orange lines seem to coincide.

This is also reflected in the second plot of Fig. 9, where the agreement of two or more tools (red, green and purple area) decreases steadily from block 4.5 M (bin 45) onwards, while the shares of contracts flagged exclusively by Conkas (pink), eThor (blue) or Mythril (gray) increase, such that at the end of the timeline, the four groups are roughly of the same size.

Our explanation for the disagreement between the tools for this weakness as well as for most others, is the lack of commonly agreed, unambiguous definitions, which is backed by our work on a unified ground truth (di Angelo and Salzer, 2023). On the surface, the tools aim for the same weakness, motivated by similar examples, but the respective interpretations and implementations may differ considerably.

Fig. 9 SWC-107 Reentrancy on a timeline of blocks, in bins of 100 k blocks. Top: Percentage of bytecodes flagged, per tool. Upper middle: Percentage of overlaps. Lower middle: Error rate of tools. Bottom: Failure rate of tools

11.2 Assessment of Tools

Based on the results of our evaluation, we summarize the observed properties of the tools.

Conkas. With an average runtime of 119 s and 4 GB of memory, Conkas belongs to the light-weight tools. The number of contracts timing out or running out of memory is small. It seems that Conkas underwent a not entirely successful update for the operations of recent forks, as the source code seems to support them, but the tool fails on contracts using them, resulting in a high number of program exceptions. This may be caused by a divergence between Rattle, the module generating the intermediate representation, and the analysis module on top. Despite these problems, Conkas reports many findings.

Ethainter and MadMax. These two tools are among the most efficient and robust tools. With average runtimes of 71 s and 21 s, respectively, and 4 GB of memory, they are fast and never exceed the allotted memory. The few errors reported are timeouts under control of the tools. The number of failures is zero, indicating a high engineering quality. This may be due to the robust base component Soufflé and the use of Datalog as a high-level specification language.

eThor. With an average runtime of 574 s, a large number of timeouts, and 369 bytecodes running out of memory even with 32 GB, this tool is one of the elephants in our study. The high resource consumption may be caused by the complex workflow – eThor is the only tool trying to show the absence of a weakness. The use of the strongly-typed programming language Java explains the absence of type errors (as we see with Python programs). eThor seems to support the most frequent operations introduced by forks, but throws errors for some of them. This makes the decreasing rate of reported reentrancy issues (and the increasing rate of contracts found secure) an unreliable indicator for the assumption that the frequency of reentrancy weaknesses indeed drops.

Maian. With an average runtime of 48 s and 4 GB memory sufficing for almost all bytecodes, Maian is a lightweight. As it is the oldest tool and unmaintained, it supports hardly any of the newer operations. This results in the second highest number of errors (reporting unknown opcodes) and virtually no weakness detections for newer contracts.

Mythril. With an average runtime of 670 s, Mythril belongs to the slow tools, but almost never needs more than 4 GB of memory. It is the only actively maintained tool in our collection: Every issue we reported was fixed within a few days. Mythril supports all EVM operations and checks for a large number of weaknesses. This, and the tendency to report also issues of low severity, result in the third largest number of flagged contracts.

Oyente and Osiris. With an average runtime of 35 s and 165 s, respectively, the two tools are among the faster tools. Osiris extends Oyente and checks for further properties, which explains the additional time it takes. 4 GB of memory suffice for most contracts. However, Oyente runs out of 32 GB of memory for about 2 000 bytecodes, whereas Osiris seems to require less memory and hardly ever exceeds the quota. Both tools fail for operations beyond fork 4.37 M, with Osiris issuing a message and Oyente failing silently. Consequently, both tools report no weaknesses for recent contracts.

Pakala. With an average runtime of 1115 s, this tool is by far the slowest, which seems to be a consequence of Pakala performing symbolic execution without optimizations. The tool author aimed at a small and simple program and deliberately omitted techniques like the construction of control flow graphs. [12] In spite of being able to handle all relevant operations, Pakala flags only 4 232 contracts as vulnerable, which may have different causes. First, the analysis of 80 000 contracts timed out, so a prolonged analysis might have revealed further

[12] https://www.palkeo.com/en/projets/ethereum/pakala.html

weaknesses. Second, Pakala might actually spend the extra time for a more refined analysis, leading to a lower number of false positives. Third, the simplicity of the program might have resulted in a lower detection rate. To determine the actual cause, we would need to check the quality of the results, which is beyond the scope of our study.

Securify. The average runtime of 160s makes Securify one of the faster tools, even though it times out for 1651 bytecodes. None of the runs exceeds the memory quota. The implementation language Java prevents type errors, but we see almost 10 000 null pointer exceptions. Even though Securify has been superseded by a successor (that supports source code only and thus does not fall into the scope of our study) and is unmaintained now, it supports most essential EVM operations. Nevertheless, its detection rate starts to drop early on and falls to virtually zero towards the end.

teEther. The average runtime of 572 s is comparable to Mythril, but teEther times out in 52 250 cases (compared to 2 620 for Mythril). The tool is exceptional regarding its appetite for memory: even with 32 GB provided, 40 306 analyses exceed the memory quota. teEther addresses a single vulnerability that it detects in 3 230 bytecodes. The tool supports the essential EVM operations, but is unmaintained now. With 6 608 Python exceptions, teEther seems to be an experimental tool focusing on the elaborate analysis of a single issue.

Vandal. With an average runtime of 63 s and 4 GB of memory, Vandal is one of the fast and light tools, but still runs into a timeout for 2 662 bytecodes and exceeds 32 GB of memory for another 1 142. The number of 1 047 programming issues is moderate for a tool written in Python. With 75% of the contracts flagged, Vandal surpasses the detection rate of the other tools. The high rate triggered some plausibility checks in Section 8.1, showing that most bytecodes containing a `CALL` operation are flagged as containing an unchecked or reentrant call. Vandal seems to implement rather unspecific criteria that lead to a large number of false positives. This interpretation is supported by Vandal's repository, where the patterns for weakness detection are listed just as use cases for a framework that decompiles bytecode to single static assignments. The accompanying paper, on the other hand, presents Vandal as a tool for vulnerability detection.

11.3 Combining or Comparing Tools Results

When comparing or combining tool results, we face two challenges: (i) different aims of tools that are reflected in the way their findings are reported and (ii) differing definitions of weaknesses (that are associated with the findings), which makes it hard to map a finding to a class (within a common frame of reference) for comparison or combination.

The tools can be divided into four groups with respect to their aim (for a specific weakness): (i) proving the absence of a property that is regarded as a weakness or vulnerability, (ii) over-reporting as to not overlook a potential weakness (aka issuing warnings), (iii) under-reporting since only those weaknesses are reported where a verification could be found (avoiding false alarms), (iv) reporting properties that are hardly a weakness (e.g. honeypots) or not necessarily (e.g. gas issues).

This distinction is important when comparing tools. It strongly affects the number of agreements. As we have seen in Sections 10 and 11.1, the overall agreement is low, which is partly due to the fact that tools address different versions, subsets or supersets of a weakness class. Considering the different aims of the tools, the low general agreement is not surprising. However, it is even low for tools with similar aims.

The aims of the tools also impact voting schemes that combine the results of several tools to 'determine' whether a contract is actually vulnerable. For over-reporting tools, it may

make sense to have a majority vote. However, under-reporting tools should rather be joined than intersected.

11.4 Comparison to Source Code as Input

We selected the tools in our study for their ability to process runtime code, as our goal was to analyze contracts deployed on the mainchain, for which Solidity source code is often unavailable. Moreover, this allowed us to include Ethainter, eThor, MadMax, Pakala, teEther and Vandal, which require runtime code. The other selected tools accept both, bytecode and Solidity source code. In this section, we discuss the effect of using source code as the input.

Conkas, Osiris, Oyente and Securify compile the Solidity source to runtime code and then perform the same analysis as if the latter had been the input. There are two differences, though. First, the tools are able to report the location of weaknesses within the source, as they use a mapping provided by the compiler to translate bytecode addresses back to line numbers. Second, for Solidity sources with more than one contract, the tools compile and analyze each one separately. As complex contracts are structured into several layers of intermediate contracts using inheritance, this leads to redundant work. While compilation and address mapping incur a negligible overhead, the additional contracts may lead to fewer or more findings within a fixed time budget, depending on whether there is less time for the main contract or whether other contracts contribute additional findings.[13]

Maian and Mythril compile the Solidity source as well but proceed with the deployment code, which includes contract initialization as well. Maian deploys the contract on a local chain and checks some properties live, like whether the contract accepts Ether. Moreover, the findings are filtered for false positives by trying to exploit the contract on the chain. Mythril, on the other hand, uses the deployment code to analyze also the constructor. For both tools, resource requirements and results will vary with the chosen form of input.

11.5 Threats to Validity

Internal validity is threatened by integrating the new tools into SmartBugs. We mitigated this threat by carefully following the SmartBugs instructions for tool integration and by consulting the documentation and the source code of the respective tools. Multiple authors manually analyzed all execution errors to ensure that we had configured the tools adequately. Moreover, we make the implementation and the results accessible for public inspection.

External validity is threatened by the use of single bytecodes as proxies for code families identified by the same skeleton. These representatives may not accurately reflect the code properties of all family members that are relevant to weakness detection. We mitigated this threat by the first research question. However, the random sample of 1 000 bytecodes (620 code families) may have been chosen too small such that our answer to RQ1 may not generalize to all bytecodes.

The focus on runtime bytecode as the sole object of analysis restricts the number of tools usable for our study, as well as the methods applicable. Some trends and observations may thus not generalize to smart contract analysis in general.

Construct validity is threatened by our mapping of the detected weaknesses to the classes of the SWC registry. The mapping reflects our understanding of the weaknesses and what the

[13] An easy remedy would be to extend the tools by a parameter with the name of the contract to analyze.

tools actually detect, which may be incorrect. We mitigated this risk by involving all authors during the mapping phase and by discussing disagreements until we reached a consensus.

Another potential threat are the resources, 30 minutes and up to 32 GB per tool and bytecode. This configuration is in line with related work or surpasses it.

12 Related Work

12.1 Recent Systematic Reviews on Analysis Tools

Two studies from early 2022 show that the automated analysis of Ethereum smart contracts has still room for improvement. Rameder et al. (2022) describe the functionalities and methods of 140 tools (83 open source) for automated vulnerability analysis of Ethereum smart contracts. Their literature review identifies 54 vulnerabilities, with some not addressed by any of the tools. Moreover, the authors find many tools to be unmaintained. Kushwaha et al. (2022) provide a systematic review of 86 analysis tools with a focus on 13 common vulnerabilities. For quality assessment, they select 16 tools, which they test on five vulnerabilities using a ground truth of 30 contracts.

12.2 Tool Evaluations without Test Sets

In 2019, two surveys evaluate tools for vulnerability detection by installing them and working through the documentation: di Angelo and Salzer (2019) investigated 27 tools with respect to availability, maturity, methods employed, and security issues detected. López Vivar et al. (2020) evaluated 18 tools regarding the ease of installation, usefulness, and updates. Both studies do not assess the detection capabilities of the examined tools.

12.3 Benchmarked Evaluations

Most closely related to our work are evaluations of tools that actually test them against a set of contracts (benchmark set). When tool authors compare their own artifact to a few similar and/or popular ones, we consider those works to be intrinsically biased and therefore do not include them.

Among the independent evaluations, we find 11 related works (Dika, 2017; Parizi et al., 2018; Gupta, 2019; Durieux et al., 2020; Ghaleb and Pattabiraman, 2020; Leid et al., 2020; Zhang et al., 2020; Dias et al., 2021; Ji et al., 2021; Ren et al., 2021; Kushwaha et al., 2022) of which we give an overview in Table 13. In the first two rows, we indicate the respective reference and the year when the evaluation was carried out. Rows three to five list the size of the benchmark set, separated into vulnerable and non-vulnerable contracts, or unknown number of vulnerable contracts. All references use Solidity files as benchmarks. Row six indicates the number of different vulnerabilities tested. We highlight low numbers in red and commendable high numbers in green. We also list for each tool which evaluation it was part of. We highlight the five tools most often used in light blue. In the last row, there is the total number of tools used in each study. We highlight the five references using the most tools in mid-blue.

Table 13 Overview of Evaluations with Benchmarks

Reference	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	Our study
Evaluation Year	2017	2018	2019		2020			2021				2022
Contracts: vulnerable	23	10	162	69	9 369	39	176	94	237	214		
Contracts: non-vulnerable	21							128				
Contracts: unknown				47 518		20				45 622	30	248 238
# Vulnerabilities	4	12+	16	10	7	9	49	57	7	8	5	15
Conkas												✓
ContractFuzzer										✓		
Echidna						✓						
Ethainter												✓
eThor												✓
ILF										✓		
HoneyBadger				✓							✓	
MadMax												✓
Maian				✓			✓				✓	✓
Manticore				✓	✓	✓			✓		✓	
Mythril		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Osiris				✓			✓			✓	✓	✓
Oyente	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓
Pakala												✓
RA											✓	
Remix-IDE	✓		✓				✓					
Securify	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
sFuzz										✓	✓	
Slither			✓	✓	✓		✓	✓	✓	✓	✓	
SmartCheck	✓	✓	✓	✓	✓		✓		✓	✓	✓	
SODA											✓	
SolHint									✓		✓	
teEther												✓
SolidityCheck							✓				✓	
Vandal												✓
VeriSmart									✓		✓	
VeriSolid											✓	
# Tools	4	4	6	9	6	3	9	3	8	9	16	12

The earliest evaluation was (Dika, 2017), which covers four tools tested on five vulnerabilities with a benchmark set of 23 vulnerable and 21 non-vulnerable contracts. Regarding the benchmark sets, the number of contracts contained shows a large variety from only 10 to almost 50 000. The number of vulnerable contracts in the benchmark set also varies largely from 10 to 9369 [14]. The number of different vulnerabilities varies from 4 to 57. Several

[14] It should be noted that for the *wild* benchmark sets, i.e. from the contracts actually deployed on the mainchain, the true number of vulnerable contracts and the vulnerabilities they contain is yet unknown.

Table 14 Contributions and Focus of Evaluations with Benchmarks

	Reference	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	Our study
Contribution	systematic review(SLR)											✓	
	weakness classification	✓		✓				✓	✓				
	test set of contracts			✓	✓	✓		✓					✓
	framework				✓	✓				✓			✓
	principles, methods					✓					✓		✓
Focus	state-of-the-art	✓										✓	
	tool effectiveness		✓	✓	✓	✓	✓	✓	✓	✓			
	tool settings										✓		
	creation of test set					✓							✓
	timeline												✓

evaluations use their own taxonomy of vulnerabilities. This may be due to the lack of an established taxonomy (Rameder et al., 2022).

We find a total of 20 tools mentioned in the evaluations, while each work selects a subset thereof for its tests. The number of tools tested varies from three to a maximum of 16. The tools most often included in a comparison are Mythril, Oyente, Securify, Slither, and SmartCheck.

In Table 14, we give an overview of the main contributions and the focus of the benchmarked evaluations. The contributions include a systematic literature review (Kushwaha et al., 2022), their own classification scheme for vulnerabilities or weaknesses (Dika, 2017; Gupta et al., 2020; Zhang et al., 2020; Dias et al., 2021), a new or newly assessed benchmark set of contracts (Gupta et al., 2020; Durieux et al., 2020; Ghaleb and Pattabiraman, 2020; Zhang et al., 2020), a framework for tool execution or test case generation (Durieux et al., 2020; Ghaleb and Pattabiraman, 2020; Ji et al., 2021), a quantitative tool evaluation (all benchmarked evaluations), or new principles and methods for tool evaluations. Regarding principles and methods, Ghaleb and Pattabiraman (2020) demonstrate the automated generation of vulnerable contracts by injecting buggy coding patterns, while Ren et al. (2021) evaluate settings for tool executions.

As for the focus, most studies address the effectiveness of tools in detecting weaknesses in smart contracts, two studies strive for a general review (Dika, 2017; Kushwaha et al., 2022), while Ren et al. (2021) aim for insights into the influence of parameter settings onto tool results.

12.4 Differences to the Benchmarked Studies

Compared to the benchmarked studies mentioned above, our study stands out in the following aspects.

Focus. Our study is the only one to focus on the *temporal evolution* of weaknesses and tool behavior as well as on reducing the number of necessary test cases while maintaining

full coverage of the Ethereum mainchain. Regarding tool effectiveness, we deliberately do not address it per se – due to the lack of suitable benchmark sets as the available ones are either small, biased, outdated, or inconsistent (di Angelo and Salzer, 2023). Rather, we are striving for a relative comparison of tools with regard to two aspects: tool behavior over time and against tools that address sufficiently similar weaknesses (via mapping to a common frame of reference).

Input. We use *runtime bytecode* as input, while the other studies use Solidity source code.

Tools. We include *further tools* like Conkas, Ethainter, eThor, MadMax, Pakala, teEther, and Vandal. As they accept bytecode only as input, neither of them was used in any of the other studies.

Size. With a benchmark set of 248328 unique contracts from the main chain, we use the *largest number of contracts*.

Contribution. Our study features a *novel method for selecting a benchmark set* that allows for analyzing an entire ecosystem, the extension of an *execution framework to work with bytecode as input*, and the inclusion of *time as a further dimension* to look at weaknesses and reasons for tool behavior.

12.5 Open Source Frameworks

For a large-scale evaluation, we need an analysis framework that (i) facilitates the control of multiple tools via a uniform interface, (ii) allows for bulk operation, and (iii) is open source and usable. SmartBugs (Ferreira et al., 2020) is such an execution framework released in 2019. It is still being maintained with 13 contributors and over 70 resolved issues. The framework USCV (Ji et al., 2021) implemented similar ideas in mid-2020. It comprises an overlapping set of tools and an extension of the ground truth set. With a total of 10 commits (the latest in mid-2021) and no issues filed, it seems to be neither widely used nor maintained. Both frameworks target Solidity source code, and thus need to be expanded to work with bytecode.

13 Conclusion

In this work, we investigated the evolution of smart contract weaknesses as reported by analysis tools. We managed to cover all 48 million smart contracts deployed on the Ethereum main chain up to block 14 000 000, by selecting tools that are able to process runtime bytecode, and by choosing only one representative for each group of contracts with the same skeleton. In total, we ran 12 tools on 248 328 contracts with a cumulative execution time of 30 years. We summarize our contributions and observations.

Skeletons are an effective technique to identify similar contracts. Clustering contracts by their skeleton reduces the computing effort as well as the bias that is introduced when sampling contracts from a population that contains some contracts once and others thousandfold. We show that the validity of studies like ours is not affected by picking only one contract per cluster.

The rate of reported weaknesses decreases over time. The tools report a total of 1 307 484 weaknesses, the most common ones being *Reentrancy* (14%), *Unchecked Call Return Value* (14.0%) and *Integer Overflow and Underflow* (9.4%). The weaknesses are not equally

distributed over the study period, though. By and large, we observe for all tools and all weaknesses a decrease in flagged contracts over time. We offer three explanations.

Some tools are no longer maintained and cannot handle operations added to the EVM later on. As such operations get more widely used, the tools increasingly fail in their analyses.

Even tools interpreting all operations correctly, may detect weaknesses by code patterns that are tied to specific compiler versions. As the code generator changes with newer compilers, the code patterns become less effective in indicating the weakness.

But the decrease in flagged contracts can also be observed for maintained and recent tools, which indicates that the weaknesses become indeed less prevalent over time. This may be attributed to factors like the adoption of good programming practices, public repositories with tested code, enhancements to the programming language Solidity, and checks added by the Solidity compiler.

The analysis tools differ considerably regarding resource consumption and engineering aspects. We see large differences in average runtimes, in the number of analyses timing out or running out of memory, and in the number of errors and failures. These aspects are of relevance in practice, e.g. when integrating analysis tools in CI/CD workflows.

The tools agree only partially in their judgment of contracts, with the disagreement increasing over time. Our overlap analysis shows that tools targeting the same weakness flag rather different sets of contracts. The intersection of these sets decreases over time. We attribute this phenomenon to diverging interpretations of the weaknesses, as precise and commonly accepted definitions are lacking. Regarding the change over time, our data provides no explanation.

Service to the community. In the course of our study, we found several bugs in tools, which we reported either by filing issues or by exchanging emails and engaging in discussions. The extension of SmartBugs to process bytecode has already been taken up by the framework Centaur[15].

Recommendations to smart contract developers, tool authors, and the community at large. From the experience gathered in this study, we derive the following recommendations and wishes.

- When hardening or auditing smart contracts, use a range of analysis tools, as their approaches and abilities are complementary. Grant the tools sufficient resources, memory- and timewise.
- Maintain academic tools for some years after publishing the accompanying article, and keep them public, as a service to the community. This allows researchers to evaluate new methods against the state of the art, on recent data.
- Strive for an abstract definition of the weakness addressed, using e.g. some formal semantics, execution traces, and path conditions. Give a precise definition of the code patterns used to detect the weakness. This makes it easier to analyze the scope of tools and to interpret their results.
- Work towards a comprehensive, balanced ground truth. Ultimately, many interesting questions regarding the quality of tools and their methods can only be answered by having access to an 'oracle' saying true or false. This goal is interlinked with the previous one, as the latter determines the meaning of the former.

[15] https://github.com/mchara01/centaur

Appendix

Table 15 Mapping of Tool Findings to SWC Classes

Tool	Finding	SWC Class
Conkas	Integer_Overflow	101
Conkas	Integer_Underflow	101
Conkas	Reentrancy	107
Conkas	Time_Manipulation	116
Conkas	Transaction_Ordering_Dependence	114
Conkas	Unchecked_Low_Level_Call	104
Ethainter	AccessibleSelfdestruct	106
Ethainter	TaintedDelegatecall	112
Ethainter	TaintedOwnerVariable	124
Ethainter	TaintedSelfdestruct	105
Ethainter	TaintedStoreIndex	124
Ethainter	TaintedValueSend	105
eThor	insecure	107
MadMax	OverflowLoopIterator	101
MadMax	UnboundedMassOp	128
MadMax	WalletGriefing	113
Maian	Destructible	106
Maian	Ether_leak	105
Mythril	Delegatecall_to_user_supplied_address_SWC_112	112
Mythril	Dependence_on_predictable_environment_variable_SWC_116	116
Mythril	Dependence_on_predictable_environment_variable_SWC_120	120
Mythril	Dependence_on_tx_origin_SWC_115	115
Mythril	Exception_State_SWC_110	110
Mythril	External_Call_To_User_Supplied_Address_SWC_107	107
Mythril	Integer_Arithmetic_Bugs_SWC_101	101
Mythril	Jump_to_an_arbitrary_instruction_SWC_127	127
Mythril	Multiple_Calls_in_a_Single_Transaction_SWC_113	113
Mythril	State_access_after_external_call_SWC_107	107
Mythril	Unchecked_return_value_from_external_call_SWC_104	104
Mythril	Unprotected_Ether_Withdrawal_SWC_105	105
Mythril	Unprotected_Selfdestruct_SWC_106	106
Mythril	Write_to_an_arbitrary_storage_location_SWC_124	124
Osiris	Concurrency_bug	114
Osiris	Overflow_bugs	101
Osiris	Reentrancy_bug	107
Osiris	Timedependency_bug	116
Osiris	Underflow_bugs	101
Oyente	Re_Entrancy_Vulnerability	107
Oyente	Timestamp_Dependency	116

Table 15 continued

Tool	Finding	SWC Class
Oyente	`Transaction_Ordering_Dependence_TOD`	114
Pakala	`call_bug`	105
Pakala	`delegatecall_bug`	112
Pakala	`selfdestruct_bug`	105
Securify	`DAO`	107
Securify	`DAOConstantGas`	107
Securify	`TODAmount`	114
Securify	`TODReceiver`	114
Securify	`TODTransfer`	114
Securify	`UnhandledException`	104
Securify	`UnrestrictedEtherFlow`	105
teEther	`Ether_leak`	105
Vandal	`Destroyable`	106
Vandal	`OriginUsed`	115
Vandal	`ReentrantCall`	107
Vandal	`UncheckedCall`	104
Vandal	`UnsecuredValueSend`	105

Acknowledgements This project was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project UIDB/50021/2020. The project was also partially supported by the CASTOR Software Research Centre. The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Program.

Funding Open access funding provided by TU Wien (TUW).

Data and Code Availability The data and the scripts of our study are available from https://figshare.com/s/5efef6335fa98ddc3ae2. The dataset of 248 328 contracts with distinct skeletons has additionally been published at https://github.com/gsalzer/skelcodes. SmartBugs is developed as a GitHub project at https://github.com/smartbugs/smartbugs. Some utilities for the manipulation of bytecode, like the computation of skeletons, are maintained at https://github.com/gsalzer/ethutils.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

di Angelo M, Salzer G (2019) A Survey of Tools for Analyzing Ethereum Smart Contracts, IEEE international conference on decentralized applications and infrastructures (DAPPCON), pp 69–78. Piscataway, NJ, USA. https://doi.org/10.1109/DAPPCON.2019.00018

di Angelo M, Salzer G (2024) Consolidation of ground truth sets for weakness detection in smart contracts. In: Essex A, Matsuo S, Kulyk O, Gudgeon L, Klages-Mundt A, Perez D, Werner S, Bracciali A, Goodell G (eds) Financial Cryptography and Data Security. FC 2023 International Workshops, Springer, LNCS, pp 439–455, https://doi.org/10.1007/978-3-031-48806-1_28

Brent L, Jurisevic A, Kong M, Liu E, Gauthier F, Gramoli V, Holz R, Scholz B (2018) Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv https://doi.org/10.48550/arXiv.1809.03981

Brent L, Grech N, Lagouvardos S, Scholz B, Smaragdakis Y (2020) Ethainter: a smart contract security analyzer for composite vulnerabilities, Association for Computing Machinery. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, London UK, PLDI 2020 16:454–469, New York, NY, USA. https://doi.org/10.1145/3385412.3385990

Chen H, Pendleton M, Njilla L, Xu S (2020) A Survey on Ethereum Systems Security. ACM Comput Surv 53(3):1–43. https://doi.org/10.1145/3391195

Dias B, Ivaki N, Laranjeiro N (2021) An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools, IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC), p 17–26. IEEE. https://doi.org/10.1109/PRDC53464.2021.00013

Dika A (2017) Ethereum Smart Contracts: Security Vulnerabilities and Security Tools. NTNU,

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, New York, NY, USA. p 530–541. ACM https://doi.org/10.1145/3377811.3380364,

Ferreira JF, Cruz P, Durieux T, Abreu R (2020) Smartbugs: A framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, p 1349–1352, ACM, New York, NY, USA, https://doi.org/10.1145/3324884.3415298,

Ferreira Torres C, Schütte J, State R (2018) Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. Proceedings of the 34th Annual Computer Security Applications Conference, pp 664–676, New York, NY, USA https://doi.org/10.1145/3274694.3274737,

Ghaleb A, Pattabiraman K (2020) How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM New York, NY, USA, pp 415–427. https://doi.org/10.1145/3395363.3397385

Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y (2018) MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. Proceedings of the ACM on Programming Languages, ACM New York, NY, USA, 2(OOPSLA):1–27. https://doi.org/10.1145/3276486,

Gupta BC (2019) Analysis of Ethereum Smart Contracts - A Security Perspective. Indian Institute of Technology Kanpur

Gupta BC, Kumar N, Handa A, Shukla SK (2020) An Insecurity Study of Ethereum Smart Contracts. In: Batina L Picek S Mondal M (eds) Security Privacy, Cryptography Applied . Springer International Publishing, Cham, Engineering, pp 188–207

Ji S, Kim D, Im H (2021) Evaluating Countermeasures for Verifying the Integrity of Ethereum Smart Contract Applications. IEEE Access, 9:90029–90042, IEEE https://doi.org/10.1109/ACCESS.2021.3091317,

Krupp J, Rossow C, (2018) teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In: 27th USENIX conference on security symposium (USENIX Security 18), Baltimore, MD USENIX Association, (18):1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

Kushwaha SS, Joshi S, Singh D, Kaur M, Lee H-N (2022) Ethereum Smart Contract Analysis Tools: A Systematic Review. IEEE Access. https://doi.org/10.1109/ACCESS.2022.3169902

Kushwaha SS, Joshi S, Singh D, Kaur M, Lee H-N (2022) Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. IEEE Access 10:6605–6621. https://doi.org/10.1109/ACCESS.2021.3140091

Leid A, van der Merwe B, Visser W (2020) Testing Ethereum Smart Contracts: A Comparison of Symbolic Analysis and Fuzz Testing Tools. In: Conference of the South African Institute of Computer Scientists and Information Technologists 2020. ACM New York, NY, USA, pp 35–43. https://doi.org/10.1145/3410886.3410907,

López Vivar A, Castedo AT, Sandoval Orozco AL, García Villalba LJ (2020) An analysis of smart contracts security threats alongside existing solutions. Entropy 22(2):203. https://doi.org/10.3390/e22020203

Luu L, Chu D-H, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, ACM New York, NY, USA, pp 254–269 https://doi.org/10.1145/2976749.2978309,

Mueller B (2018) Smashing ethereum smart contracts for fun and real profit. 9th Annual HITB Security Conference (HITBSecConf). Amsterdam, Netherlands HITB, https://raw.githubusercontent.com/b-mueller/smashing-smart-contracts/master/smashing-smart-contracts-1of1.pdf,

Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A (2018) Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th annual computer security applications conference. New York, NY, USA ACM. pp 653–663. https://doi.org/10.1145/3274694.3274743,

Parizi RM, Dehghantanha A, Choo Kim-Kwang R, Singh A (2018) Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: Proceedings of the 28th annual international conference on computer science and software engineering. vol 18 pp 103–113, IBM Corp. http://dl.acm.org/citation.cfm?id=3291291.3291303,

Rameder H, di Angelo M, Salzer G (2022) Review of automated vulnerability analysis of smart contracts on ethereum. Front Blockchain 5. https://doi.org/10.3389/fbloc.2022.814977

Ren M, Yin Z, Ma F, Xu Z, Jiang Y, Sun C, Li H, Cai Y (2021) Empirical evaluation of smart contract testing: what is the best choice? In: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis. pp 566–579. ACM New York, NY, USA. https://doi.org/10.1145/3460319.3464837

Schneidewind C, Grishchenko I, Scherer M, Maffei M (2020) EThor: practical and provably sound static analysis of ethereum smart contracts. Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. Association for Computing Machinery, New York, NY, USA. pp 621–640. https://doi.org/10.1145/3372297.3417250

Tang X, Zhou K, Cheng J, Li H, Yuan Y (2021) The vulnerabilities in smart contracts: a survey. In: Sun X, Zhang X, Xia Z, Bertino E (eds) International conference on artificial intelligence and security (ICAIS). Communications in computer and information science, vol CCIS 1424, Springer, Cham, pp 177–190. https://doi.org/10.1007/978-3-030-78621-2_14

Tolmach P, Li Y, Lin S-W, Liu Y, Li Z (2022) A survey of smart contract formal specification and verification. ACM Comput Surv 54(7):1–38. https://doi.org/10.1145/3464421

Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M (2018) Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp 67–82. ACM New York, NY, USA. https://doi.org/10.1145/3243734.3243780

Wang Z, Jin H, Dai W, Choo K-KR, Zou D (2021) Ethereum smart contract security research: survey and future research opportunities. Front Comput Sci 15(2):152802. https://doi.org/10.1007/s11704-020-9284-9

Zhang P, Xiao F, Luo X (2020) A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), pp 139–150. https://doi.org/10.1109/ICSME46990.2020.00023

Zhou H, Milani Fard A, Makanju A (2022) The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. J Cybersec Priv, 2(2):358–378. Multidisciplinary Digital Publishing Institute, https://doi.org/10.3390/jcp2020019

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Monika di Angelo[1] · **Thomas Durieux**[2] · **João F. Ferreira**[3] · **Gernot Salzer**[1]

Thomas Durieux
thomas@durieux.me

João F. Ferreira
joao@joaoff.com

Gernot Salzer
gernot.salzer@tuwien.ac.at

1 TU Wien and INESC-ID Lisbon, Vienna, Austria

2 TU Delft, Delft, Netherlands

3 INESC-ID and Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal