frontiers in Blockchain

REVIEW
published: 24 March 2022
doi: 10.3389/fbloc.2022.814977

Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum

Heidelinde Rameder, Monika di Angelo and Gernot Salzer*

Faculty of Informatics, TU Wien, Vienna, Austria

Programs on public blockchains often handle valuable assets, making them attractive targets for attack. At the same time, it is challenging to design correct blockchain applications. Checking code for potential vulnerabilities is a viable option to increase trust. Therefore, numerous methods and tools have been proposed with the intention to support developers and analysts in detecting code vulnerabilities. Moreover, publications keep emerging with different focus, scope, and quality, making it difficult to keep up with the field and to identify relevant trends. Thus, regular reviews are essential to keep pace with the varied developments in a structured manner. Regarding blockchain programs, Ethereum is the platform most widely used and best documented. Moreover, applications based on Ethereum are entrusted with billions of USD. Like on similar blockchains, they are subject to numerous attacks and losses due to vulnerabilities that exist at all levels of the ecosystem. Countermeasures are in great demand. In this work, we perform a systematic literature review (SLR) to assess the state of the art regarding automated vulnerability analysis of smart contracts on Ethereum with a focus on classifications of vulnerabilities, detection methods, security analysis tools, and benchmarks for the assessment of tools. Our initial search of the major on-line libraries yields more than 1,300 publications. For the review, we apply a clear strategy and protocol to assure consequent, comprehensive, and reproducible documentation and results. After collecting the initial results, cleaning up references, removing duplicates and applying the inclusion and exclusion criteria, we retain 303 publications that include 214 primary studies, 70 surveys and 19 SLRs. For quality appraisal, we assess their intrinsic quality (derived from the reputation of the publication venue) as well as their contextual quality (determined by rating predefined criteria). For about 200 publications with at least a medium score, we extract the vulnerabilities, methods, and tools addressed, among other data. In a second step, we synthesize and structure the data into a classification of both the smart contract weaknesses and the analysis methods. Furthermore, we give an overview of tools and benchmarks used to evaluate tools. Finally, we provide a detailed discussion.

Keywords: systematic literature review, taxonomy, security, tools, vulnerability, analysis, benchmark

OPEN ACCESS

Edited by:
Giovanni Meroni,
Politecnico di Milano, Italy

Reviewed by:
Raimundas Matulevicius,
University of Tartu, Estonia
Claudio Di Ciccio,
Sapienza University of Rome, Italy

****Correspondence:***
Monika di Angelo
monika.di.angelo@tuwien.ac.at

Specialty section:
This article was submitted to Smart Contracts, a section of the journal Frontiers in Blockchain

Received: *14 November 2021*
Accepted: *24 January 2022*
Published: *24 March 2022*

Citation:
Rameder H, di Angelo M and Salzer G (2022) Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. Front. Blockchain 5:814977. doi: 10.3389/fbloc.2022.814977

1 INTRODUCTION

Smart contracts, as blockchain programs are called, extend blockchains from a platform for financial transactions to an all-purpose utility. With their distinguishing features observability, tamper evidence, and automatic enforcement, they are the protagonists of trustless computation. As such, they promise to advance application domains like supply chain management, medical services, and especially (decentralized) finance. However, Wang Z. et al. (2020) warn that "any successful breach, particularly those that are highly publicized, can impact the community's belief in smart contracts, and hence its usage."

Tolmach et al. (2021) observe that "the adoption of blockchains and smart contracts is also accompanied by severe attacks, often due to domain-specific security pitfalls in the smart contract implementations." As one of the reasons for vulnerabilities, Singh et al. (2020) state that "writing secure and safe smart contracts can be extremely difficult due to various business logics, as well as platform vulnerabilities and limitations". Furthermore, Vacca et al. (2020) point out that "smart contracts and blockchain applications are developed through non-standard software life-cycles, in which, for instance, delivered applications can hardly be updated or bugs resolved by releasing a new version of the software."

Currently, Ethereum is the major platform for decentralized applications (dApps) and decentralized finance (DeFi). Its ecosystem consists of the underlying blockchain, a large variety of smart contracts deployed on it, a wide range of valuable assets (most notably fungible and non-fungible tokens managed by smart contracts), and the Ethereum Foundation, which coordinates the efforts of an enthusiastic community and of supporting companies like ConsenSys. While developers go to great lengths to avoid security issues, even experts introduce or overlook major vulnerabilities. Not surprisingly, Ethereum is confronted with high-stake attacks and losses, thus actual and potential vulnerabilities are a major concern.

This situation motivates researchers and practitioners to devise methods and tools for the development of secure smart contracts to avoid pitfalls from the outset, and for screening smart contracts for vulnerabilities. These efforts include the documentation of vulnerabilities, the collection of best practices, and the automated detection of issues. To date, more than 100 tools have been presented that either support the development of blockchain programs or help to analyze them once created. The number of publications detailing the methods and comparing the approaches seems overwhelming. Therefore, surveys aim at systematizing the contributions and summarizing the state of the art.

Among the many aspects of smart contract, our systematic literature review focuses on studies related to vulnerabilities and their automated detection. We assess the publications with respect to the following questions:

- Which vulnerabilities are mentioned? How are vulnerabilities classified?
- Which methods do automated tools use to detect vulnerabilities?
- Which vulnerabilities are addressed by the tools?
- How are the tools evaluated?
- What are the open challenges of automated detection?

Starting from an initial list of 1300+ related publications, we apply a carefully documented process to select and evaluate 303 studies of immediate relevance, and to distill current trends and open challenges. Our contributions are

- a list of high quality publications, with the selection based on clear criteria,
- an overview and taxonomy of vulnerabilities, methods and tools,
- a discussion of the state of the art with respect to automated analysis, and
- the identification of gaps and challenges.

The paper is structured as follows. **Section 2** describes the search for and the selection of 303 publications, the criteria for quality appraisal, and the structure of the publications found. In **Sections 3–7**, we give a synthesis of the extracted data. **Section 3** summarizes eight related systematic literature reviews, while **Section 4** classifies the vulnerabilities we found. **Section 5** gives an overview of the methods used in the automated analysis of smart contracts, whereas **Section 6** concentrates on the tools implementing them. Test sets and benchmarks, required for the evaluation of tools, are summarized in **Section 7**. **Section 8** is devoted to the discussion of our findings, before we conclude with **Section 9**.

The extensive **Supplementary Material** contains a description of all vulnerabilities, organized in a consolidated taxonomy, an overview of the 140 tools we found, and the quality appraisal or surveys and primary studies.

2 SYSTEMATIC REVIEW

For the literature review, we apply a rigorous protocol to ensure comprehensive and reproducible results as suggested in several guidelines (Brereton et al., 2007; Kitchenham and Charters, 2007; Okoli, 2015; Snyder, 2019). This section documents the initial search, the first selection and classification, as well as the quality appraisal. In subsequent sections, we discuss the extracted data in form of a synthesis, focusing on the aspects: related work (**Section 3**), vulnerability taxonomies (**Section 4**), analysis methods (**Section 5**), tools (**Section 6**), and benchmark sets (**Section 7**).

2.1 Search and Selection

2.1.1 Initial Search

In accordance with our research questions, we choose the primary search term *smart contract* and a disjunction of the terms *security*, *vulnerability*, *bug*, *analysis*, *detection*, *verification* and *tool*. One of the lessons learned in (Brereton et al., 2007) is the necessity to query several databases, as in the domain of information systems, there is not a single source containing all relevant papers. The initial search was conducted at the end of January 2021 and

TABLE 1 | Query strings on different search engines.

Database	Query
ACM DL	Title: ("smart contract" OR "smart contracts") AND AllField: (vulnerability OR vulnerabilities OR bug OR bugs OR tool OR security OR analysis OR detection OR verification)
Google Scholar	allintitle: ("smart contract" OR "smart contracts" OR Ethereum) AND (vulnerability OR vulnerabilities OR bug OR bugs OR security OR analysis OR detection OR tool OR verification)
IEEE Xplore	"All Metadata": ("smart contract" OR "smart contracts") AND (vulnerability OR vulnerabilities OR bug OR bugs OR tool) AND (security OR analysis OR detection OR verification)
Science Direct	Title, abstract, keywords: "smart contract" AND (vulnerability OR vulnerabilities OR bug OR tool OR security OR analysis OR detection OR verification)
TU Wien CatalogPlus	title contains ("smart contract" OR "smart contracts") AND Subject contains (security OR vulnerability OR vulnerabilities OR bug OR bugs Or analysis OR detection OR tool OR verification)

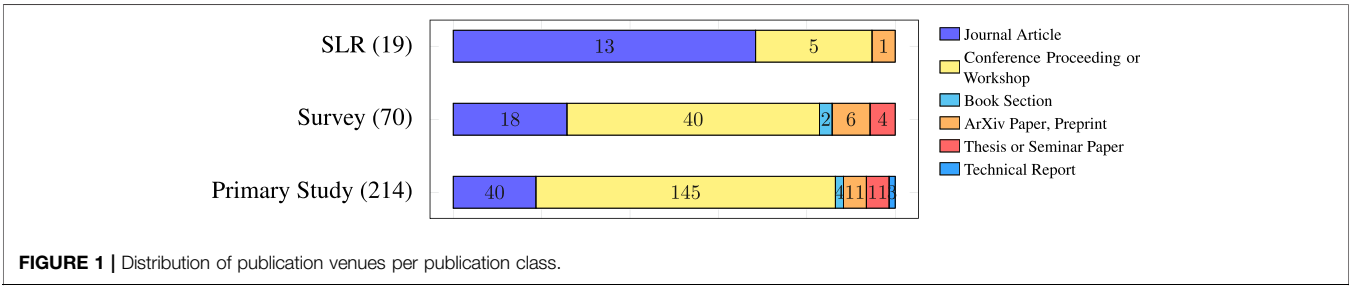

FIGURE 1 | Distribution of publication venues per publication class.

yielded a total of 1326 results, with 546 contributed by Google Scholar, 229 by IEEE Xplore, 199 by ScienceDirect, 178 by TU Wien CatalogPlus, and 174 by ACM Digital Library.

Table 1 lists the database queries. Using *blockchain code/program* as synonyms for *smart contract* does not increase the number of results. In general, we search the titles of publications as well as their meta-information, including keywords and abstracts. For Google Scholar, we have to limit the search to titles, as an unrestricted search yields more than 50 000 hits, most unrelated to smart contracts. To make up for the brevity of titles, we add *Ethereum* as an alternative to *smart contract* in this case.

2.1.2 Exclusion and Inclusion Criteria

Exclusion Criteria: We exclude search results if the work is 1) not written in English, 2) a patent, blog entry or other grey literature, 3) not accessible on-line via the library network of TU Wien, 4) published before 2014, and 5) unrelated to Ethereum.

The last exclusion criterion is debatable for an academic review, as it restricts our attention to a specific technology. However, within the area of smart contracts, Ethereum clearly dominates the public blockchains with respect to the diversity of applications, the market cap, and the size of its community. An increasing number of entrepreneurs and visionaries propose new business models, which prompts developers to design languages and to implement tool chains for this platform. The financial values involved make Ethereum an attractive target for attackers, leading in turn to research on vulnerabilities and counter-measures. This self-reinforcing spiral lead to a situation where the amount of literature on Ethereum smart contracts vastly exceeds the one on other platforms, which seems to justify our focus.

Inclusion Criteria: We include search results if it is clear from title and abstract that the work is related to 1) smart contract security bugs or vulnerabilities, 2) smart contract vulnerability detection, analysis or security verification tools, or 3) automated detection or verification methods.

2.1.3 Selection and Classification

After removing 299 duplicates and excluding 724 articles according to the criteria above, we group the remaining 303 studies into three classes:

- *Systematic Literature Reviews (SLRs)*,
- *Surveys* including surveys or review studies of smart contract security analysis tools, methods, approaches or vulnerabilities, and
- *Primary Studies* including research on the development of smart contract security analysis and vulnerability detection tools, methods or approaches.

Figure 1 breaks down the publications by class and venue.

2.2 Quality Appraisal

In the next step, we read the selected literature in detail. Besides extracting various data, we assess the quality of the papers. If a work does not meet minimum standards, it is excluded from the subsequent literature review (Okoli, 2015). Similarly to Varela-Vaca and Quintero (2021), we apply the intrinsic and contextual data quality metrics described in Strong et al. (1997).

2.2.1 Intrinsic Data Quality

Intrinsic data quality (IDQ) assesses the accuracy, objectivity, credibility, and reputation of the publication venue. We map the *SCImago Journal Ranking* (SCImago, 2021), the *CORE Journal or Conference Ranking* (CORE, 2021), and the *Scopus CiteScore*

TABLE 2 | Mapping from popular venue rankings to our unified intrinsic data quality (IDQ).

SCImago	CORE	Scopus CiteScore	IDQ score
Q1	A*, A	> 75 percentile	1.0
Q2	B	> 50 percentile	0.8
Q3, Q4	C	≥ 25 percentile	0.5
not indexed, but pub. by IEEE, ACM, Springer			0.4
		< 25 percentile	0.2
otherwise, e.g. arXiv, preprints, theses, reports			0.2

percentile ranking (Scopus, 2021) to a score between 0 and 1 (**Table 2**). In the case of multiple rankings we take the highest score as IDQ. If a journal or conference has not (yet) been ranked for the year under consideration, we consider the most recent ranking.

Figure 2 breaks down the IDQ score of the 303 initially selected publications per publication class. A total of 126 publications (41.6%) is rated at the highest IDQ score of 1.0, while 44 publications (14.5%) are rated at 0.8 and 27 (8.9%) at 0.5. At the lower end, we find 56 publications (18.5%) with a score of 0.4 and 50 (16.5%) with a score of 0.2.

Figure 3 shows the IDQ scores within the three publication classes. Most SLRs (15 out of 19, 78.9%) have been published in highly reputable venues, compared to only 17 of 70 (24.3%) surveys scoring at 1.0. Overall, the quality of surveys is the lowest, with 36 (51.4%) scoring 0.4 or 0.2. Of the selected primary studies, 126 of 214, almost 60% score 0.8 or 1.0.

2.2.2 Contextual and Final Data Quality

Contextual data quality (CDQ) assesses the relevance, added value, timeliness, completeness and amount of data, and thus depends on the context of the evaluation, like the purpose of the systematic review, the research questions, and the data to extract and analyze. We devise two questionnaires, **Tables 3**, **4**, to evaluate the CDQ of primary and secondary studies. The CDQ score is the arithmetic mean of the answers to the questions, each answer being a number between 0 and 1.

$$\text{CDQ} = \frac{\text{sum of answers}}{\text{number of questions answered}}$$

The final data quality (FDQ) is a combination of IDQ and FDQ. We compute the FDQ score as the arithmetic mean of the other two scores. It is thus also a number between 0 and 1.

$$\text{FDQ} = \frac{\text{IDQ} + \text{CDQ}}{2}$$

For the sake of readability, and ease of classification, we map CDQ and FDQ to a three point Likert scale, by calling a score low if it is below 0.5, high if it is at least 0.8, and medium otherwise. We exclude a study from the survey if its CDQ or FDQ is low. The rationale for considering CDQ twice, once explicitly and once as part of the FDQ score, is to retain work of medium CDQ that has not been published yet but is available from public repositories (resulting in a low IDQ and FDQ score). We could achieve a similar effect by weighting CDQ in the computation of the FDQ score.

2.2.3 Quality Appraisal of Systematic Literature Reviews

The questionnaire for measuring CDQ (**Table 3**) is based on the DARE criteria suggested in (Kitchenham and Charters, 2007). The quality appraisal for the systematic literature reviews (19 initially and one subsequently identified) is presented in **Table 5**. The table includes the IDQ score, the answers of the CDQ questionnaire, CDQ and FDQ score, and the corresponding CDQ and FDQ values. Of the 20 reviews, we reject nine due

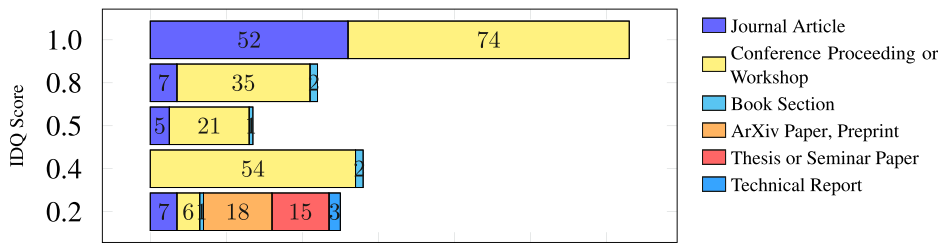

FIGURE 2 | Number of publications per IDQ score, by publication venue.

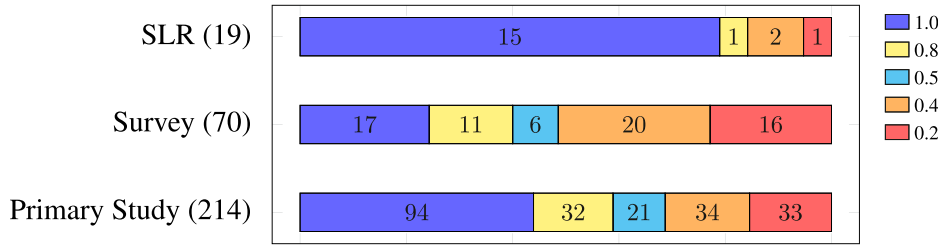

FIGURE 3 | Distribution of IDQ scores per publication class.

TABLE 3 | Criteria for assessing the contextual data quality (CDQ) of SLRs and surveys.

Q1: Is the methodology and literature search of the review systematic and documented? Are all relevant studies at the time likely to be covered?
1.0 = Yes, in comprehensive detail, systematic and good quality. To be (re)classified as SLR.
Not Graded = No, to be (re)classified as survey
Q2: Does the study focus on work about smart contact vulnerabilities, detection tools, or approaches to automated detection or verification?
1.0 = Yes, the research topic closely related and covers major parts of this work
0.6 = It covers related topics, but important areas differ or are missing
0.3 = Some similar areas are covered
0.0 = The review focuses on different research topics, only a very small part is related
Q3: Does the study cover the evaluation of smart contract vulnerabilities?
1.0 = Yes, the work includes a survey, evaluation and detailed description of vulnerabilities including classifications
0.6 = Descriptions or classifications of certain vulnerabilities are given, but it is not a major focus of the work
0.3 = Some vulnerabilities are superficially listed or mentioned
0.0 = No, the focus is not on the survey of vulnerabilities
Q4: How many tools or analysis/verification methods does the review identify, classify, compare or evaluate?
1.0 = more than 12, 0.6 = 8 to 12, 0.3 = 4 to 7, 0.0 = less than 4
Q5: Did the author(s) assess and compare the quality/validity of tools or automated detection/verification approaches in detail?
1.0 = Yes, in detail and high quality, including practical experiments or comprehensive/in-depth coverage and classification
0.6 = Yes, compared/evaluated in some detail, but only theoretically
0.3 = Only superficial description
0.0 = No
Q6: Is a set of smart contract benchmarks provided?
1.0 = Yes, Not Graded = No
Q7: How current is the review?
1.0 = 2020/21, 0.6 = 2019, 0.3 = 2018, 0.0 = 2017 and older

TABLE 4 | Criteria for assessing the contextual data quality (CDQ) of primary studies.

Q1: Is the primary study referenced in a selected SLR or survey? How often is it cited according to Google Scholar?
1.0 = referenced in selected SLR or Survey, or cited more than 12 times according to Google Scholar
0.6 = 8 to 12 citations, 0.3 = 4 to 7 citations, 0.0 = less than 4 citations
Q2: Is the primary study related to an executable tool or to a framework for verifying security properties or identifying vulnerabilities of Ethereum smart contracts?
1.0 = Yes, 0.0 = No

to low CDQ or FDQ and reclassify three as surveys. Eight reviews, summarized in **Section 3**, remain for data extraction.

2.2.4 Quality Appraisal of Surveys

The **Supplementary Material** of this article contains a table similar to **Table 5** that assesses the quality of 70 initially selected surveys, four additional surveys subsequently identified, and three publications reclassified from SLR to survey. **Figure 4** gives an overview. Of the 77 publications, we reclassify six as primary study and remove two duplicates. We reject 31 surveys due to low CDQ or FDQ and retain 38 for data extraction.

2.2.5 Quality Appraisal of Primary Studies

The assessment of primary studies focuses on the identification of tools for the automated analysis of security issues (see the questionnaire in **Table 4**). We consider 224 primary studies, including six studies initially classified as survey and four studies identified only subsequently. Based on the quality appraisal in the **Supplementary Material**, we reject 75 publications because of low CDQ or FDQ, and retain 149 for further data extraction (**Figure 5**).

2.3 Data Extraction

After quality appraisal and reclassification, we are left with 8 systematic literature reviews, 38 surveys and 149 primary studies for data extraction. **Figure 6** aggregates the number of studies by the year of publication. Apparently, the research in the field grows rapidly, with the primary studies jumping from 3 in 2017 to 40 in 2018 and the number of SLRs and surveys nearly tripling from 2019 to 2020. Next, in **Sections 3–7**, we give a synthesis of the extracted data.

3 RELATED WORK

In this section, we summarize the eight SLRs of high contextual or final data quality that we identified in our initial search. We list them in chronological order and discuss their relation to our work.

3.1 Summary of SLRs With High Quality

Liu and Liu (2019) focus on smart contract verification and select 53 publications, with 20 addressing security assurance and 33 correctness verification. For security assurance, the authors identify the three categories environment security, vulnerability scanning and performance impacts, whereas correctness verification is subdivided into program correctness and formal verification. The paper mentions various tools in each category, but does not compare them

TABLE 5 | Quality appraisal of systematic literature reviews. Q6 does not apply to this type of publications.

systematic literature review	IDQ	CDQ								FDQ		Selected
	score	Q1	Q2	Q3	Q4	Q5	Q7	score	value	score	value	
Ante (2021)	1.0	1.0	0.3	0.0	0.3	0.0	1.0	0.43	low	0.72	med	✘
Chen et al. (2020a)	1.0	reclassified as survey										
Coblenz et al. (2019)	1.0		0.0	0.0	0.0	0.0	0.6	0.12	low	0.56	med	✘
Guo et al. (2021)	1.0	1.0	0.0	0.0	0.0	0.0	1.0	0.33	low	0.67	med	✘
Gupta et al. (2020)	0.8	reclassified as survey										
Hu et al. (2021) [a]	0.2	1.0	1.0	0.3	1.0	0.6	1.0	0.82	high	0.51	med	✓
Kim and Ryu (2020)	0.4	1.0	0.6	0.6	1.0	0.6	1.0	0.80	high	0.60	med	✓
Leka et al. (2019)	0.4		0.3	0.3	0.0	0.0	0.6	0.24	low	0.32	low	✘
Liu and Liu (2019)	1.0	1.0	0.6	0.0	0.6	0.3	0.6	0.52	med	0.76	med	✓
Macrinici et al. (2018)	1.0	1.0	0.3	0.6	0.0	0.0	0.3	0.37	low	0.68	med	✘
Rouhani and Deters (2019)	1.0	1.0	0.3	0.3	0.6	0.3	0.6	0.52	med	0.76	med	✓
Singh et al. (2020)	1.0	1.0	1.0	0.0	1.0	0.6	1.0	0.77	med	0.88	high	✓
Sanchez-Gómez et al. (2020)	1.0	1.0	0.3	0.0	0.0	0.0	1.0	0.38	low	0.69	med	✘
Taylor et al. (2020)	1.0	1.0	0.3	0.0	0.0	0.0	1.0	0.38	low	0.69	med	✘
Tolmach et al. (2020)	0.2	1.0	0.6	0.6	1.0	0.6	1.0	0.80	high	0.50	med	✓
Tovanich et al. (2019)	1.0	1.0	0.0	0.0	0.0	0.0	0.6	0.27	low	0.63	med	✘
Vacca et al. (2020)	1.0	1.0	0.6	0.0	1.0	0.6	1.0	0.70	med	0.85	high	✓
Varela-Vaca and Quintero (2021)	1.0	1.0	0.3	0.0	0.0	0.0	1.0	0.38	low	0.69	med	✘
Zeli Wang et al. (2020)	1.0	1.0	0.6	0.6	0.6	0.3	1.0	0.68	med	0.84	high	✓
Zhang et al. (2020)	1.0	reclassified as survey										

[a]*Identified subsequently.*

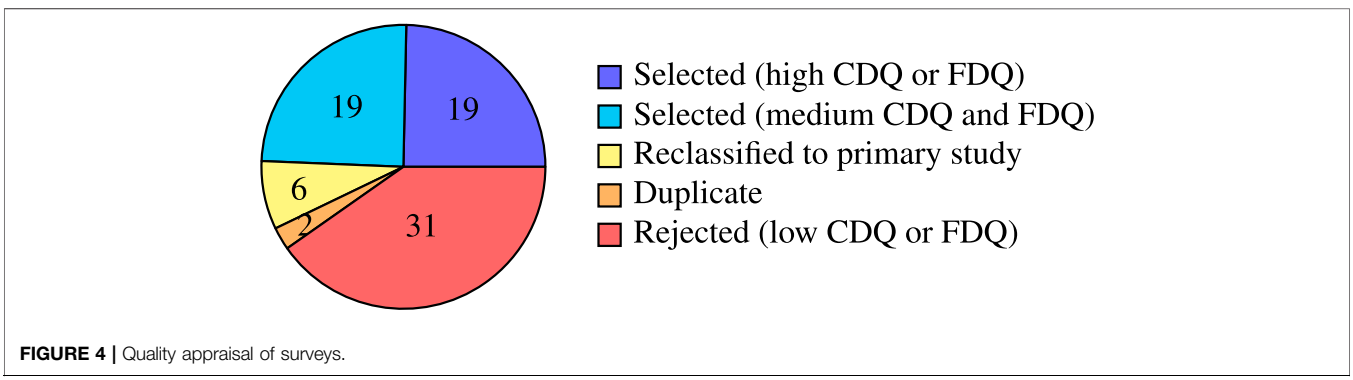

FIGURE 4 | Quality appraisal of surveys.

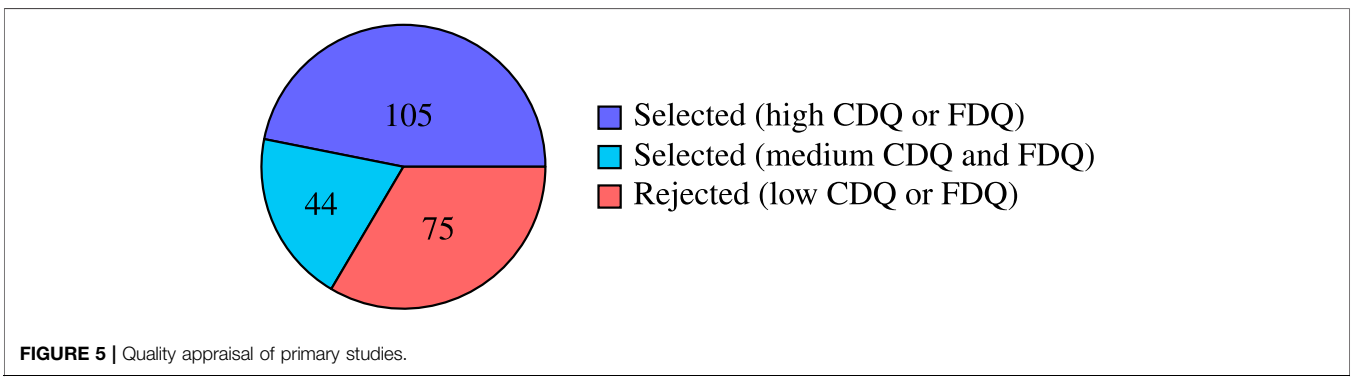

FIGURE 5 | Quality appraisal of primary studies.

directly. The authors conclude that methods for formally verifying correctness are an effective way to ensure smart contract credibility and accuracy.

Rouhani and Deters (2019) conduct a systematic review regarding the security, performance and applications of smart contracts, finally considering 90 papers. They describe the

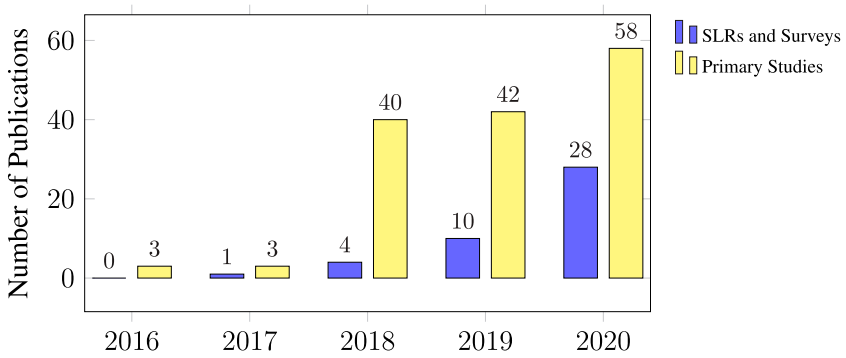

FIGURE 6 | Publication trend per year for SLRs and surveys as well as primary studies.

security issues and present methods and tools to analyze them. The authors identify four major security problems, namely transaction order dependence, timestamp dependence, mishandled exceptions, and reentrancy. They evaluate nine vulnerability analysis tools and summarize methods for formal verification as well as for the detection of effective callback free objects.

Zeli Wang et al. (2020) systematically survey the literature on the security of Ethereum smart contracts, published between July 2015 and July 2019. The main contributions relevant to our work are a systematic mapping and a taxonomy of security problems including counter-measures, with the three major categories "abnormal contract", "program vulnerability" and "exploitable habitat". The methods for detecting vulnerabilities are described at a high level, divided into static analysis (including symbolic execution and formal verification), dynamic analysis and code similarity. The authors describe individual tools, but neither perform a comprehensive evaluation nor map vulnerabilities to the detection methods.

Singh et al. (2020) analyze work published between 2015 and July 2019, synthesizing a final set of 35 research papers on formal approaches to avoid vulnerabilities in smart contracts. The most common approach is the verification of security properties by theorem proving, whereas symbolic execution and model checking are frequently used to establish functional correctness. Further formal techniques comprise formal modeling, finite state machines, logic based approaches, behavioral modeling, formal reasoning and formal specification languages. The authors provide a mapping between formal techniques and the addressed issues in smart contracts. Moreover, they identify 15 formal tools and frameworks and relate them to the formal methods used.

Tolmach et al. (2020) give a comprehensive survey of the methods for formally specifying and verifying smart contracts, based on 202 papers published from September 2014 to June 2020. They present a taxonomy of formal approaches, with the main categories being modeling formalisms, specification formalisms, and verification techniques. Modeling formalisms are contract-level models, such as process algebras, state-transition systems and set-based methods, as well as program-level models, like abstract syntax tree analysis, control-flow automata and program logics. Specification formalisms are divided into formal specifications such as contract and program-level specifications, as well as properties by domain, like security, privacy, finance, social games and asset tracking. The verification techniques comprise model checking, theorem proving, program verification, symbolic and concolic execution, runtime verification, and testing. In total, the survey describes 34 verification tools and frameworks and associates them with the respective formalism. In their conclusion, the authors state that there is still a lack in clear approaches and standards with respect to secure development and analysis techniques. Furthermore, they argue that different blockchains and smart contract platforms often require different approaches to security analysis. Recently, this survey has been published as (Tolmach et al., 2021).

Kim and Ryu (2020) give a survey of the analysis of smart contract for various blockchains, based on 67 out of 391 initial papers. Of these, 24 papers use static analysis for vulnerability detection, 24 static analysis for program correctness, and 19 use dynamic analysis. The approaches are further subdivided according to specific methods like symbolic execution, abstract interpretation, machine learning, fuzzing, runtime verification, and concolic testing. The authors evaluate 27 tools regarding the ability to detect one or more of 19 vulnerabilities. They point out unsolved challenges such as program behavior and language ambiguities, and highlight promising research directions such as the design of new languages and type systems, and the use of machine learning.

Hu et al. (2021) review papers from the period 2008–2020 that focus on design paradigms, tools for developing secure smart contracts, and systems to improve privacy or efficiency, selecting finally 159 studies and twelve related surveys. The parts most relevant to our work are the evaluation and classification of analysis methods and tools. The authors identify, describe, and classify 40 tools that are able to detect and analyze vulnerabilities. Of these, 15 focus on the detection of specific vulnerabilities, while the others have a broader scope like identifying multiple vulnerabilities, verifying custom properties, or alerting to potential security risks. Additionally, the review presents 20 auxiliary tools, including frameworks and high level languages.

In their conclusion, the authors state that many tools are inefficient and require specific knowledge for defining security properties. They also notice a trade-off between accuracy and the coverage of multiple vulnerabilities.

Vacca et al. (2020) focus on current challenges, techniques and tools for smart contract development. The survey is based on 96 articles, published between 2016 and 2020, on the analysis and the testing of smart contracts, on metrics for and security of smart contracts, on Dapp performance and on blockchain applications. The work summarizes the properties and application areas of 26 tools for the automated analysis of smart contracts. Moreover, the review describes experimental datasets and 18 empirical validations. The authors emphasize the need for guidelines and further research regarding the development and testing of smart contracts.

3.2 Relation to Our Work

The SLRs above focus on the development of smart contracts, on analysis methods, on formal methods, and on security issues. Automated analysis is covered in varying degrees, but is not at the center. Vulnerabilities are described in one review, a taxonomy is suggested by two. Most SLRs include a description of the methods found, but usually without indicating the vulnerabilities that can be tackled by the methods. Tool descriptions are more often included than not, while comparisons of tool properties are less frequent. The conclusions of the SLRs portray an immature field, in particular with respect to standards and guidelines, program behavior, tool efficiency, and testing. This situation and the marked increase in publications warrants regular reviews of the state of the art.

Naturally, our review includes more recent research, up to January 2021, as it was conducted later than the other SLRs. What sets our work apart is its specific scope, its breadth, and rigor. Our main focus is automated vulnerability detection, including tools, taxonomies and benchmarks. Starting from 1300+ initial search results, we assessed the relevance and quality of 303 publications in detail, applying clearly defined criteria (cf. **Section 2**). For the chosen scope, our review with a **Supplementary Material** of 70+ pages can be regarded as a comprehensive overview of the state-of-the-art at the beginning of 2021.

4 CLASSIFICATIONS OF VULNERABILITIES

In this section, we give an overview of classification schemes. We start with our consolidated taxonomy of the vulnerabilities identified in the body of literature. Then we summarize classifications by scholars and present two community taxonomies. Finally, we present a mapping of our consolidated taxonomy to the community classifications.

In the reviewed literature, the term *vulnerability* is used in a broader sense than is common in computer security. It refers to a weakness or limitation of a smart contract that may result in security problems. A vulnerability allows for the execution of a smart contract in unintended ways. This includes locked or stolen resources, breaches of confidentiality or data integrity, and state changes in the environment of smart contracts that were not anticipated by developers or users and that put some involved party at an advantage or disadvantage.

4.1 Consolidated Taxonomy

In total, we extracted 54 vulnerabilities from the collected papers. The **Supplementary Material** contains a short description for each, including references. Our consolidated classification in **Table 6** consists of 10 classes of vulnerabilities. It is based on 17 systematically selected surveys (as documented in the supplement) and two popular community classifications presented below.

4.2 Academic Classifications

Of the early, frequently cited papers on smart contract vulnerabilities, only some present a novel classification scheme or refine an existing one.

Luu et al. (2016) describe Ethereum smart contract vulnerabilities, such as transaction-ordering dependence (TOD), timestamp dependence and mishandled exceptions and reentrancy, without additional grouping. They define the vulnerabilities and present code snippets, examples of attacks, and affected real live smart contracts. To fix some problems, they propose improvements to the operational semantics of Ethereum, namely guarded transactions (countering TOD), deterministic timestamps and enhanced exception handling.

Atzei et al. (2017) create one of the first taxonomies for vulnerabilities. At the top, vulnerabilities are classified according to where they appear: in the source code (usually Solidity), at machine level (in the bytecode or related to instruction semantics), or at blockchain level. A mapping to actual examples of attacks and vulnerable smart contracts completes the taxonomy. Although this work is referenced in several other papers, we have found some issues and inconsistencies regarding the classification of concrete vulnerabilities. For example, the vulnerability type called *unpredictable state* is illustrated by an example that is viewed in most other work as an instance of *transaction order dependency*. At the same time another example for problems associated with *dynamic libraries* is assigned to the same class. It can be argued that these two examples exhibit different vulnerabilities, as the underlying causes are inherently different. Dika (2017) extends the taxonomy of Atzei et al. (2017) by adding further vulnerabilities and assessing the level of criticality.

Grishchenko et al. (2018) present a formal definition of the semantics of the EVM as well as of the security properties *call integrity*, *atomicity*, *independence of the transaction environment* and *independence of a mutable account state*. If a bytecode satisfies such a property, it is provably free of the corresponding vulnerabilities. As the properties usually are too complex to be established automatically, the authors consider simpler criteria that imply the properties. E.g., the safety property *Single Entrancy* implies call integrity and precludes that a contract suffers from the reentrancy vulnerability (Schneidewind et al., 2020).

4.3 Classifications by Community Based Projects

4.3.1 DASP Top 10

The Decentralized Application Security Project (DASP), initiated by the private organization NCC Group (2018), identifies ten

TABLE 6 | Consolidated taxonomy of vulnerabilities of smart contracts on Ethereum.

Code	Vulnerability
1	**Malicious Environment, Transactions or Input**
1A	Reentrancy
1B	Call to the unknown
1C	Exact balance dependency
1D	Improper data validation
1E	Vulnerable DELEGATECALL
2	**Blockchain/Environment Dependency**
2A	Timestamp dependency
2B	Transaction-ordering dependency (TOD)
2C	Bad random number generation
2D	Leakage of confidential information
2E	Unpredictable state (dynamic libraries)
2F	Blockhash dependency
3	**Exception & Error Handling Disorders**
3A	Unchecked low level call/send return values
3B	Unexpected throw or revert
3C	Mishandled out-of-gas exception
3D	Assert, require or revert violation
4	**Denial of Service**
4A	Frozen Ether
4B	Ether lost in transfer
4C	DoS with block gas limit reached
4D	DoS by exception inside loop
4E	Insufficient gas griefing
5	**Resource Consumption & Gas Issues**
5A	Gas costly loops
5B	Gas costly pattern
5C	High gas consumption of variable data type or declaration
5D	High gas consumption function type
5E	Under-priced opcodes
6	**Authentication & Access Control Vulnerabilities**
6A	Authorization via transaction origin
6B	Unauthorized accessibility due to wrong function or state variable visibility
6C	Unprotected self-destruction
6D	Unauthorized Ether withdrawal
6E	Signature based vulnerabilities
7	**Arithmetic Bugs**
7A	Integer over- or underflow
7B	Integer division
7C	Integer bugs or arithmetic issues
8	**Bad Coding and Language Specifics**
8A	Type cast
8B	Coding error
8C	Bad coding pattern
8D	Deprecated source language features
8E	Write to arbitrary storage location
8F	Use of assembly
8G	Incorrect inheritance order
8H	Variable shadowing
8I	Misleading source code
8J	Missing logic, logical errors or dead code
8K	Insecure contract upgrading
8L	Inadequate or incorrect logging or documentation
9	**Environment Configuration Issues**
9A	Short address
9B	Outdated compiler version
9C	Floating or no pragma
9D	Token API violation
9E	Ethereum update incompatibility
9F	Configuration error
10	**Eliminated/Deprecated Vulnerabilities**
10A	Callstack depth limit
10B	Uninitialized storage pointer
10C	Erroneous constructor name

groups of smart contract vulnerabilities. The project neither defines the listed vulnerabilities nor explains how the vulnerabilities were selected and ranked. Several studies like Durieux et al. (2020) use DASP Top 10 as basis, but note that the ten categories are not sufficient.

4.3.2 SWC Registry

The Smart Contract Weakness Classification Registry (SWC Registry, 2018) relates smart contract vulnerabilities to the Common Weakness Enumeration (CWE) typology (MITRE Corp, 2006) and collects test cases. Currently, the registry holds 36 vulnerabilities, with descriptions, references, suggestions for remediation and sample Solidity contracts.

4.4 Mapping of Vulnerability Classifications

Different taxonomies are difficult to map to each other when based on complementary aspects. While several taxonomies build on the early classification of Atzei et al. (2017), the extensions diverge with respect to the dimensions they consider, like the level where vulnerabilities occur (protocol layer vs. EVM vs. Solidity) or cause vs. effect of vulnerabilities. So far, none of the taxonomies has seen wide adoption. Tools without a vulnerability classification of their own usually refer to DASP or the SWC registry.

Our consolidated taxonomy is compatible with the ones of DASP and SWC. **Table 7** maps our ten classes, omitting vulnerabilities that have no counterpart in the other taxonomies. We find a correspondence for 34 vulnerabilities, while 20 vulnerabilities documented in literature remain uncovered.

The mapping is not exact in the sense that categories in the same line of the table may overlap only partially. For example, DASP 1 covers both “reentrancy” and “call to the unknown”, while SWC only mentions “reentrancy” in 107 but not “call to the unknown”, and our taxonomy lists them separately. Moreover, some categories in our taxonomy list several SWC entries or split up categories from DASP.

With only nine genuine categories and one “catch-all”, DASP is comparatively coarse. SWC covers a range of 36 vulnerabilities, but 22 of our categories are missing. Both community classifications seem inactive: SWC was last updated in March 2020, and the DASP 10 website with the first iteration of the project is dated 2018.

5 METHODS USED IN AUTOMATED ANALYSIS

In this section, we give an overview of the methods used to detect vulnerabilities in smart contracts. For other summaries, differing in breadth and depth, see the surveys (Almakhour et al., 2020; Huasan Chen et al., 2020; di Angelo and Salzer, 2019; Garfatta et al., 2021; Hu et al., 2021; Kim and Ryu, 2020; López Vivar et al., 2020; Praitheeshan et al., 2020b; Samreen and Alalfi, 2020; Singh et al., 2020; Tolmach et al., 2021).

We discuss four groups of methods: static code analysis, dynamic code analysis, formal specification and verification, and miscellany. The distinction between static analysis and formal methods is to some extent arbitrary, as the latter are mostly used in a static context. Moreover, methods like symbolic execution regularly use formal methods as a black box. A key difference is the aspiration of formal methods to be rigorous, requiring correctness and striving for completeness. In this sense abstract interpretation should be rather considered a formal method, but it resembles symbolic execution and therefore is presented there.

5.1 Static Code Analysis

Static code analysis inspects code without executing it in its regular environment. The analysis starts either from the source or the machine code of the contract. In most cases, the aim is to identify code patterns that indicate vulnerabilities. Some tools also compute input data to trigger the suspected vulnerability and check whether the attack has been effective, thereby eliminating false positives.

To put the various methods into perspective, we take a closer look at the process of compiling a program from a high-level language like Solidity to machine code (Aho et al., 2007; Grune et al., 2012). The sequence of characters first becomes a stream of lexical *tokens* (comprising e.g. the letters of an identifier). The parser transforms the linear stream of tokens into an *abstract syntax tree (AST)* and performs semantic checks. The subsequent phases receive the AST in the form of an *intermediate representation (IR)*. Now several rounds of code analysis, code optimization, and code instrumentation may take place, with the output in each round again in IR. In the final phase, the IR is transformed into code for the target machine, like the EVM in the case of Ethereum. This last step linearizes any hierarchical structures left, by arranging code fragments into a sequence and by converting control flow dependencies to jump instructions.

5.1.1 Control Flow Graphs

For code analysis, a graph representation of the code is preferable, as it provides access to the program structure and the control flow, and eliminates irrelevant details like variable naming or register allocation. Such representations are readily available when starting from source code, as AST and IR are by-products of compilation. E.g., some tools search the AST for *syntactic patterns* characteristic of vulnerable contracts. This approach is fast, but lacks accuracy if a vulnerability cannot be adequately characterized by such patterns.

Recovering a *control flow graph (CFG)* from machine code is inherently more complex. Its nodes correspond to the *basic blocks* of a program. A basic block is a sequence of instructions executed linearly one after the other, ending with the first instruction that potentially alters the flow of control, must notably conditional and unconditional jumps. Nodes are connected by a directed edge if the corresponding basic blocks may be executed one after the other. The reachability of code is difficult to determine, as indirect jumps retrieve the target address from a register or the stack, where it has been stored by an earlier computation. Backward slicing resolves many situations by tracking down the origins of the jump targets. If this fails, the analysis has the choice between over- and under-approximation, by either treating all blocks as potential successors or by ignoring the undetectable successors.

Some tools go on by transforming the CFG (and a specification of the vulnerability) to a restricted form of Horn Logic called *DataLog*, which is not computationally universal, but admits efficient reasoning algorithms (see e.g. (Soufle, 2016)).

Starting from the CFG, *decompilation* attempts to reverse also the other phases of the compilation process, with the aim to obtain source from machine code. The result is intended for manual inspection by humans, as it usually is not fully functional and does not compile.

5.1.2 Symbolic Execution

Symbolic execution is a method that executes the bytecode like a regular machine would do, but with symbols as placeholders for arbitrary input and environment data. Any operation on such symbols results in a symbolic expression that is passed to the next operation. In the case of a fork, all branches are explored, but they are annotated with complementary symbolic conditions that restrict the symbols to those values that will lead to the execution of the particular branch. At intervals, an *SMT (Satisfiability Modulo Theory) solver* is invoked to check whether the constraints on the current path are still simultaneously satisfiable. If they are contradictory, the path does not correspond to an actual execution trace and can be skipped. Otherwise, exploration continues. When symbolic execution reaches code that matches a vulnerability pattern, a potential vulnerability is reported. If, in addition, the SMT solver succeeds in computing a satisfying assignment for the constraints on the path, it can be used to devise an exploit that verifies the existence of the vulnerability.

The effectiveness of symbolic execution is limited by several factors. First, the number of paths grows exponentially with depth, so the analysis has to stop at a certain point. Second, some aspects of the machine are difficult to model precisely, like the relationship between storage and memory cells, or complex operations like hash functions. Third, SMT solvers are limited to certain types of constraints, and even for these, the evaluation may time out instead of detecting (un)satisfiability.

Concolic execution interleaves symbolic execution with phases where the program is run with concrete input (*concolic* = *conc*rete + symb*olic*). Symbolic execution of the same path then yields formal constraints characterizing the path. After negating some constraint, the SMT solver searches for a satisfying assignment. Using it as the input for the next cycle leads, by construction, to

TABLE 7 | Mapping of classifications for vulnerabilities.

Class	Vulnerabilty	DASP	SWC	Consolidated
1	Reentrancy	1	107	1A
	Call to the unknown	1		1B
	Strict balance equality		132	1C
	Untrusted delegatecall	2	112	1E
2	Timestamp dependence	8	116	2A
	Transaction order dependence	7	114	2B
	Bad random number gen	6	120	2C
	Confidential info leak		136	2D
	Block hash dependency		120	2F
3	Unchecked return value	4	104	3A
	DOS with failed call	5	113	3B
	Gasless send		134	3C
	Assert/require violation		110, 123	3D
4	DOS block gas limit	5	128	4C
	Insufficient gas griefing	5	126	4E
6	Auth. with tx.origin	2	115	6A
	Wrong visibility	2	100, 108	6B
	Unprotected selfdestruct	5	106	6C
	Unprotected send/withdraw		105	6D
	Signature issues		117, 121, 122, 133	6E
7	Over/underflow	3	101	7A
8	Coding error		129	8B
	Deprecated Solidity		111	8D
	Arbitrary storage write		124	8E
	Arbitrary jump		127	8F
	Incorrect inheritance order		125	8G
	Variable shadowing		119	8H
	malicious code		130	8I
	logical error or dead code		131	8J
9	Short address	9		9A
	Outdated compiler version		102	9B
	Floating pragma		103	9C
10	Uninit. memory, storage		109	10B
	Constructor		118	10C

the exploration of a new path. This way, concolic execution achieves a better coverage of the code.

Taint analysis marks values from the input, the environment or from storage with tags ("taints"). Propagation rules define how tags are transformed by the instructions. Some vulnerabilities can be identified by inspecting the tags arriving at specific code locations. Taint analysis is often used in combination with other methods, like symbolic execution.

5.1.3 Abstract Interpretation

Most static methods for vulnerability detection are neither sound nor complete. They may report vulnerabilities where there are none (false positives, unsoundness), and may fail to detect vulnerabilities present in the code (false negatives, incompleteness). The first limitation arises from the inability to specify necessary conditions for the presence of vulnerabilities that can be effectively checked. The second one is a consequence of the infeasibly large number of computation paths to explore, and the difficulty to come up with sufficient conditions that can be checked.

Abstract interpretation (Cousot and Cousot, 2004) aims at completeness by focusing on properties that can be evaluated for all execution traces. As an example, abstract interpretation may split the integer range into the three groups zero, positive, and negative values. Instead of using symbolic expressions to capture the precise result of instructions, abstract interpretation reasons about how the property of belonging to one of the three groups propagates with each instruction. This way it may be possible to show that the divisors in the code always belong to the positive group, ruling out division by zero, for any input. The challenge is to come up with a property that is strong enough to entail the absence of a particular vulnerability, but weak enough to allow for the exploration of the search space. Contrary to symbolic execution and most other methods, this approach does not indicate the presence of a vulnerability, but proves that a contract is definitely free from a certain vulnerability (safety guarantee).

5.2 Dynamic Code Analysis

Dynamic code analysis checks the behavior of code, while it processes data in its "natural" environment. The most common method is testing, where the code is run with selected inputs and its output is compared to the expected result.

Fuzzing is a technique that runs a program with a large number of randomized inputs, in order to provoke crashes or otherwise unexpected behavior.

Code instrumentation augments the program with additional instructions that check for abnormal behavior or monitor performance during runtime. An attempt to exploit a vulnerability then may trigger an exception and terminate execution. As an example, a program could be systematically extended by assertions ensuring that arithmetic operations do not cause an overflow.

Machine instrumentation is similar to code instrumentation, but adds the additional checks on machine level, enforcing them for all contracts. E.g., an extended EVM might check for overflows at every arithmetic operation. Some authors go even further by proposing changes to the transaction semantics or the Ethereum protocol, in order to prevent vulnerabilities. While interesting from a conceptual point of view, such proposals are difficult to realize, as they require a hard fork affecting also the contracts already deployed.

Mutation testing is a technique that evaluates the quality of test suites. The source code of a program is subjected to small syntactic changes, known as mutations, which mimic common errors in software development. For example, a mutation might change a mathematical operator or negate a logical condition. If a test suite is able to detect such artificial mistakes, it is more likely that it also finds real programming errors.

5.3 Formal Specification and Verification

Programmers prefer high level programming languages over assembly, as it allows them to express the program logic in a more abstract way, reducing the rate of errors and speeding up development. Modeling smart contracts on an even higher level of abstraction offers additional benefits, like formal proofs of contract properties. The core logic of many blockchain applications can be modeled as *finite state machines (FSMs)*, with constraints guarding the transitions. As FSMs are simple formal objects, techniques like *model checking* can be used to verify properties specified in variants of computation tree logic. Once the model is finished, tools translate the FSM to conventional source code, where additional functionality can be added.

The high cost of errors and the small size of blockchain programs makes them a promising target for formal verification approaches. Unlike testing, which detects the presence of bugs, formal verification aims at proving the absence of bugs and vulnerabilities. As a necessary prerequisite, the execution environment and the semantics of the programming language or the machine need to be formalized. Then functional and security properties can be added, expressed in some *specification language*. Finally, automated theorem provers or semi-automatic proof assistants can be used to show that the given program satisfies the properties.

*F** is a functional programming language with a proof assistant for program verification. Bhargavan et al. (2016) develop a F* framework that translates both, Solidity source code and EVM bytecode, to F* in order to verify high- and low-level properties. Grishchenko et al. (2018) use F* to specify the small-step semantics of the EVM.

KEVM is a formal specification of the EVM in the specification language K (Hildenbrandt et al., 2018). From the specification, the K framework is able to generate tools like interpreters and model-checkers, but also deductive program verifiers.

Horn logic is a restricted form of first-order logic, but still computationally universal. It forms the basis of logic-oriented programming and is attractive as a specification language, as Horn formulas can be read as if-then rules.

5.4 Miscellany

Like in many other areas, methods from *machine learning* gain popularity also in smart contract analysis. Techniques like long-short term memory (LSTM) modeling, convolution neural networks or N-gram language models may achieve high test accuracy. A common challenge is to obtain a labeled training set that is large enough and of sufficient quality.

5.5 Usage of Methods in Tools

Figure 7 shows the number of tools using one of the methods above. Formal reasoning and constraint solving is most frequently employed, due to the many tools integrating formal methods as a black box, like constraint solvers to prune the search space or Datalog reasoners to check intermediate representations. Proper formal verification, automated or via proof assistants, is rare, even though smart contracts, due to their limited size and the value at stake, seem to be a promising application domain. This may be due to the specific knowledge required for this approach.

Next in popularity are the construction of control flow graphs (46, 32.9%), symbolic execution (44, 31.4%), and the use of intermediate representations and specification languages (37, 26.4%).

6 TOOLS FOR AUTOMATED SECURITY ANALYSIS

The 24 SLRs and surveys as well as the 149 primary studies we finally selected describe a total of 140 tools for analyzing the security of Ethereum smart contracts, with 83 published open source. In the **Supplementary Material**, we describe the tools and list their functionalities and methods.

6.1 Functionalities

Figure 8 shows the prevalence of the functionalities provided by the tools.

Code level. More than half of the tools analyze Solidity code (86, 61.4%) or bytecode (73, 52.1%), while a few (19, 13.6%) work at both levels.

Aim. Next to the detection of vulnerabilities, security analysis comprises the verification of correctness of code, gas/resource analysis, decompilation and disassembly, and correct-by-design approaches. More than half of the tools (79, 56.4%) aim at detecting

vulnerabilities, almost a third (42, 30.0%) is concerned with proving the absence of vulnerabilities, and 17 (12.1%) analyze resources.

Interaction. Some tools go the extra length of verifying the vulnerabilities they found by providing exploits or suggesting remedies. Almost a third (41, 29.2%) allows for full automation or bulk analysis, while 27 (19.2%) are aimed at manual analysis or development support.

Analysis Type. The vast majority of tools (113, 80.7%) employs static methods, about a third (43, 30.7%) uses dynamic methods, and several (16, 11.4%) use both.

6.2 Publication Trends

Figure 9 shows a break-down of the tools by the year of publication. The development of new tools has increased rapidly since 2018, with more than half of them published open source. Over a third of the open source tools (25) received updates in 2020, while 19 tools were updated within the first 7 months of 2021.

6.3 Maintenance of Tools

Tools that are not open source are difficult to assess, hence we only consider open source tools when taking a closer look at their maintenance status.

Many tools were developed as a proof-of-concept for a method described in a scientific publication, and have not been maintained since their release. While this is common practice in academia, potential users prefer tools that are maintained and where reported issues are addressed in a timely manner.

Table 8 lists twenty tools that have been around for some time, are maintained, and are apparently used. More precisely, we include a tool if it was released in 2019 or earlier, shows continuous update activities, and has some filed issues that were addressed by the authors of the tool. We exclude newer tools, since they have not yet a substantial maintenance history.

7 BENCHMARKS AND TEST SETS

In this section, we give an overview of the benchmarks and test sets for smart contract analysis that we identified in SLRs, surveys and primary studies. For each of them, **Table 9** lists the publication, the project name and a link to a repository (if available). The number of contracts per test set (fourth column) varies between 6 and 47 541. More than half of the projects also include contracts that were manually verified to be true or false positives with respect to some property, in order to serve as a ground truth. Their number is given in the fifth column. Flags indicate the availability of analysis results, source code, bytecode, and deployment addresses on Ethereum's main chain. Three collections additionally provide exploits, marked in the last column.

The first group in the table refers to collections of vulnerable contracts written in Solidity. They are not associated with any tool and provide neither analysis results, nor corresponding bytecodes, nor deployment addresses for Ethereum's main chain.

The second group in this table comprises projects aimed at the analysis of vulnerabilities, but without accompanying tool. They partially provide Ethereum addresses, Solidity sources, and analysis results.

The third and largest group consists of tools that provide test data. Most provide Solidity sources and/or Ethereum addresses, four of them also bytecode. Two tools, VerX and Zeus, offer only analysis results, but neither the bytecode, the source code nor the deployment address of the contracts analyzed, which makes it hard to verify the results.

8 DISCUSSION

In this section, we present our observations regarding the vulnerabilities, methods, tools, and benchmarks we found in the literature.

8.1 Vulnerabilities

8.1.1 Coverage

We collected 54 vulnerabilities documented in literature. While reentrancy is an early and well analyzed vulnerability, most others have received significantly less attention. This uneven coverage is also reflected in the tools, which address vulnerabilities in diverse combinations, with reentrancy being the most prominent one.

8.1.2 Taxonomies

The Ethereum ecosystem still lacks a coordinated process for collecting and structuring vulnerabilities. Collaborative efforts like the SWC registry are valuable resources, but as plain collections lack structural information and usability. We find several proposals from the community and from scholars regarding Ethereum-specific taxonomies, none of which can be considered established. Despite the narrow scope, blockchain and Ethereum, we do not perceive a convergence of taxonomies. In fact, the proposed taxonomies often are complementary rather than extensions or refinements. This makes it difficult to map the different taxonomies to each other and leaves room for discussion.

One reason may be the continued rapid development on all levels, including blockchain protocols and blockchain programming. Another one are the different angles for categorizing vulnerabilities. For detection, it is natural to consider the causes of vulnerabilities, as this is what tools can search for, like storage locations accessible to anyone. A second dimension are the effects of vulnerabilities, like denial of service or unauthorized withdrawal. Different causes can result in the same effect, while a technical cause may contribute to various effects. A third perspective looks at the motives of a potential attacker, like economic incentives or the demonstration of skill and power, relevant e.g. when trying to assess the severity of vulnerabilities. Authors of tools have their own perspective; the employed methods determine how vulnerabilities are defined and related.

Taxonomies mixing cause, effect and attacker intentions may be comprehensive, but are difficult to use when the aim is, for instance, to compare tools or to match suitable test sets with tools, as the vulnerabilities cannot be clearly assigned. A hierarchical, multi-level classification without overlaps, on the other hand, may be too strict

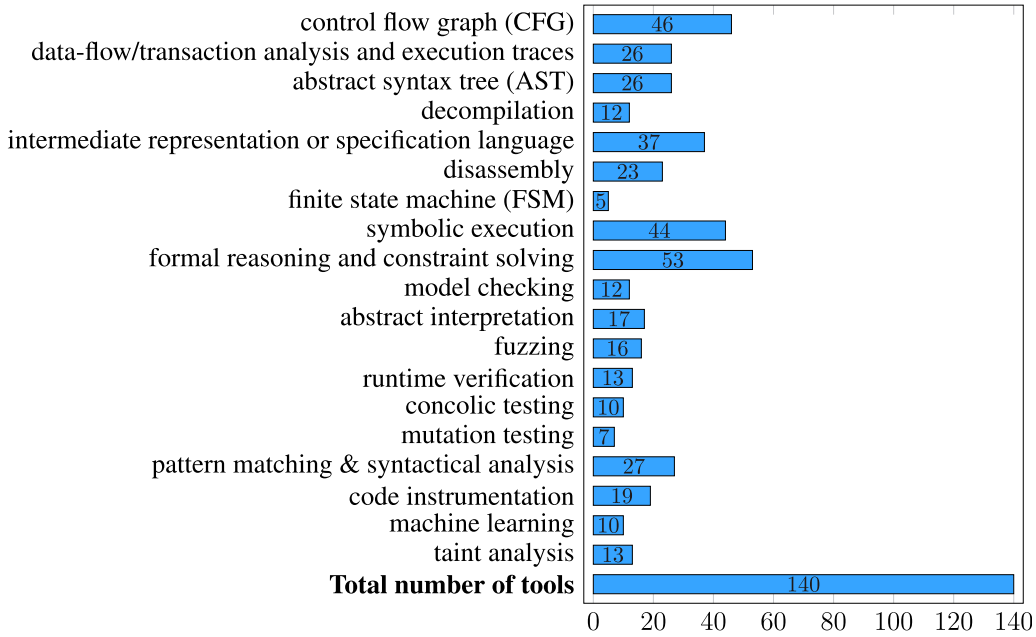

FIGURE 7 | Number of analysis tools employing a particular method.

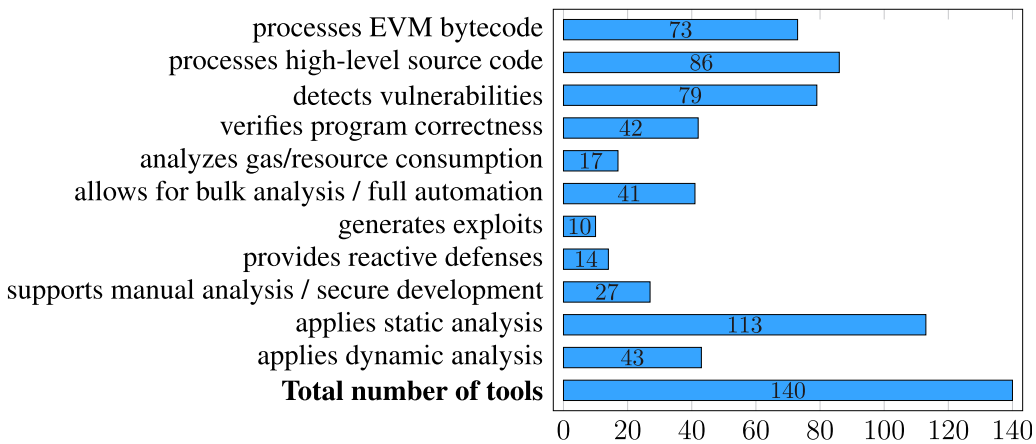

FIGURE 8 | Number of analysis tools providing a particular functionality.

to cater for multi-faceted vulnerabilities. Altogether, we see the need for more work on differentiating and systematizing vulnerabilities as well as on assessing their severity.

8.2 Methods

Searching for vulnerabilities, in source code or on machine level, is not limited to blockchains. Static and dynamic program analysis are as old as programming, and most methods we found are well-established in program analysis at large. Early on, researchers on program analysis demonstrated that methods like symbolic execution are able to detect vulnerabilities of smart contracts and to generate exploits. What makes blockchain programs a particularly attractive domain, is their limited size and the drastic consequences bugs may have. The former results in search spaces that are small compared e.g. to those of desktop programs, which makes approaches feasible that rely on the exploration of the entire search space. Thus, the application of known methods within the specific context of Ethereum may also lead to new insights and refinements outside.

Researchers from the blockchain community, on the other hand, occasionally present prototypes for their approaches that disregard the state of the art in program analysis. This is not always apparent from the publication, where the authors may state in a side note that their algorithm works on control flow graphs or extracts the entry points of the contract as starting

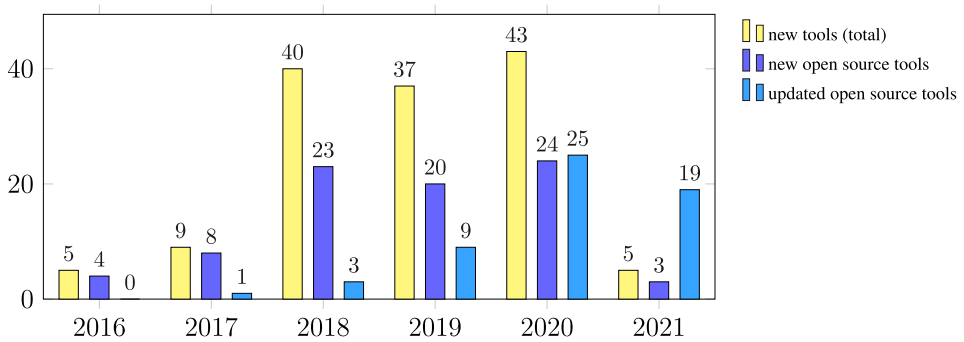

FIGURE 9 | Publication and maintenance of tools. The numbers for 2021 include the first 7 months only.

TABLE 8 | Tools published in or before 2019 that are maintained and in use.

Tool	Published	Public repository at github.com
ContractLarva	2019–08	gordonpace/contractLarva
Ethersplay	2018–05	crytic/ethersplay
Gigahorse	2019–01	nevillegrech/gigahorse-toolchain
ILF	2019–11	eth-sri/ilf
KEVM	2019–07	kframework/evm-semantics
KEVM Verifier	2018–10	runtimeverification/verified-smart-contracts
MadMax	2018–09	nevillegrech/MadMax
Manticore	2017–02	trailofbits/manticore
Mythril	2017–10	ConsenSys/mythril
Oyente	2016–01	enzymefinance/oyente
PASO	2019–09	aphd/paso
Remix-IDE	2014–11	ethereum/remix-project
Securify	2018–09	eth-sri/securify2
Slither	2018–10	crytic/slither
SmartBugs	2019–10	smartbugs/smartbugs
SmartCheck	2018–05	smartdec/smartcheck
SOLC-VERIFY	2019–07	SRI-CSL/solidity/blob/0.7/SOLC-VERIFY-README.md
Solhint	2017–10	protofire/solhint
teEther	2019–02	nescio007/teether
Vertigo	2019–08	JoranHonig/vertigo

point. Only when checking the code it becomes apparent whether the tool is a proof-of-concept with heuristics tied e.g. to a particular version of the Solidity compiler, or whether the tool performs thorough code analysis.

Publications surveying the methods used by the tools, classify them along familiar lines, like static vs. dynamic analysis, probabilistic/heuristic vs. deductive analysis, but hardly go beyond. A technical in-depth comparison of detection approaches is still lacking, as it is beyond the scope of pure surveys. It would be desirable 1) to scrutinize which methods are suited to which extent for detecting particular vulnerabilities, or inversely, 2) to determine for each vulnerability, which methods can detect it to which degree or under which conditions.

8.3 Tools

With 140 tools released until January 2021, it is difficult to decide, which tools to consider for a particular purpose.

8.3.1 Vulnerabilities Addressed

The tools address differing subsets of the 54 vulnerabilities, with most tools tackling fewer than ten. There are some frameworks that combine several tools with a unified interface to harness the power of many.

Claim vs. achievement. Each tool advertises a list of vulnerabilities that it purportedly detects. Due to the variety of methods employed, different tools may classify contracts differently, even when they seemingly address the same vulnerability. Moreover, tools may refer to incompatible taxonomies of vulnerabilities or introduce their own definition, which makes it difficult to compare the tools.

Warnings vs. confirmed vulnerabilities vs. security guarantees. Many vulnerabilities are 'detected' by rather simple heuristics. As an example, contracts are commonly reported as depending on block data, if they contain any of the instructions accessing such information. While this warning is not wrong *per se*, it does not imply that the flagged contract is

TABLE 9 | Benchmarks/test sets for smart contracts without accompanying tool.

Publication	Project/tool	Repository	no. contracts	Ground truth	Analysis results	Solidity	Bytecode	Addresses	Exploits
Trail of Bits (2020)	(Not So) Smart Contracts	url	25	25		✓			✓
SWC Registry (2018)	SWC Registry	url	122	122		✓			
–	EVM Analyzer	url	38	38		✓			
–	Ethernaut	url	21	(✓)		✓			
–	Capture the Ether	url	17	(✓)		✓			
Jiachi Chen et al. (2020)	–	url	587	587				✓	
Gupta (2019), Gupta et al. (2020)	–	–	40	40				✓	
Praitheeshan et al. (2020a)	–	url	49		✓	✓		✓	
Ye et al. (2019a),Ye et al. (2019b)	–	–	3 884	(✓)	✓	✓		✓	
Zhang et al. (2020)	JiuZhou	url	146	146		✓			
Zhou et al. (2020)	–	url	2 949		✓			✓	
Jiang et al. (2018); Mei et al. (2019)	ContractFuzzer	url	416	416		✓	✓		
Hartel and van Staalduinen (2019)	ContractVis	url	1 112		✓	✓	✓		
Haijun Wang et al. (2019), Haijun Wang et al. (2020)	ContraMaster/ Vultron	url	21		✓	✓			✓
Chen et al. (2021)	DefectChecker	url	581	581	✓		✓	✓	
Liu et al. (2018)	Ether* (S-gram)	url	1 500		✓			✓	
Kolluri et al. (2019)	ETHRACER	url	8 139	82	✓			✓	
Liu et al. (2020)	FAIRCON	url	17	17	✓	✓	✓	✓	
Yang et al. (2020)	*MSgram analysis*	url	7 124		✓	✓			
Shuai Wang et al. (2019)	NPCHECKER	url	50	50	✓			✓	
Luu et al. (2016)	OYENTE	url	17 554		✓			✓	
Albert et al. (2019)	SAFEVM	url	~ 10 000			✓		✓	
Rodler et al. (2018)	Sereum	url	6	6	✓	✓			✓
Nguyen et al. (2020)	sFuzz	url	46 186	350[a]	✓	✓		✓	
Durieux et al. (2020); Ferreira et al. (2020a),Ferreira et al. (2020a)	SmartBugs	url	47 541	143	✓	✓		✓	
Akca et al. (2019)	SolAnalyser	url	1 839			✓			
Liao et al. (2019)	SoliAudit	url	17 980		✓			✓	
Marescotti et al. (2020)	Solicitous	url	28 590			✓		✓	
Ghaleb and Pattabiraman (2020)	SolidiFI	url	350	350	✓	✓			
Zhang et al. (2019)	SolidityCheck	url	1 363		✓	✓		✓	
Hegedus (2018)	SolMet	url	10 206		✓	✓		✓	
Permenev et al. (2020)	VerX	url	13		✓				
Kalra et al. (2018)	ZEUS	url	1 524		✓				

[a]*ground truth taken from project SolidiFi.*
(✓) size of ground truth unclear

actually vulnerable, as there are safe uses of block data. Consequently, most tools are neither correct nor complete: They may report contracts as vulnerable, even though they are not, and vice versa. Since large numbers of false positives diminish trust, some tools verify their findings by constructing exploits and checking that these indeed work. Single tools choose a complementary approach by showing that the contract under consideration satisfies a security property that provably precludes a specific vulnerability.

8.3.2 Comparison and Evaluation

Apart from vulnerabilities addressed and detection methods employed, the tools differ regarding maturity and maintenance. For open source tools, the latter can be assessed by checking information on last updates and number of contributors in the respective repositories (see the **Supplementary Material** for an overview). An evaluation of tools, however, requires to install and run them on some samples, at a minimum. To our knowledge, there is no comprehensive evaluation of this type. This leads to the unsatisfactory situation that surveys and related work sections include tools that actually do not exist or are otherwise of poor quality. As an example, students envisioned *DappGuard* as part of an assignment, but did not implement it; the repository contains just a script for collecting some data from the chain. Apparently the students did an excellent job, as their technical report, accessible on the webpage of the course, is taken by many researchers as sufficient proof for the existence of the tool.

Regarding the proper evaluation of tools, we see a wide spectrum. On the one end, we find tool descriptions with a few test runs, where neither tool nor test data are accessible. Most evaluations compare the new tool to some previously published ones on selected smart contracts. The validity of such evaluations is limited, though, as the presence or absence of vulnerabilities is determined by other tools that have not been fully validated either. There are some laudable exceptions, where the authors first compile a ground truth that consists of smart contracts manually checked by one or more persons, before using it to evaluate tools.

8.4 Benchmarks

Regarding benchmarks and test sets, we find three types in literature.

- Smart contracts actually deployed, also referred to as *in the wild*, that have been obtained 1) from Ethereum's main chain (and possibly have a verified source code at Etherscan[1]), or 2) from one of the test chains.
- Contracts *generated* 1) for challenges like Ethernaut, 2) as illustration of mishaps, or 3) from a source code by introducing vulnerabilities systematically by code transformations.
- Contracts that do not possess a known vulnerability (any more). These include 1) contracts obtained from vulnerable ones by fixing the issue and that are considered touchstones, and 2) contracts appearing vulnerable to attackers, but being actually safe (honeypots).

Several test sets try to establish a ground truth for selected vulnerabilities, by confirming their existence either informally or by providing an exploit.

9 CONCLUSION

In this work, we conducted a systematic literature review regarding the automated analysis of vulnerabilities in Ethereum smart contracts. We consolidated relevant vulnerability classifications by structuring 54 vulnerabilities into ten major groups. Additionally, we compiled the properties and methods characterizing the analysis tools. We identified a total of 140 tools addressing the security of smart contracts, with 83 published open source. Finally, we gave an overview of publicly available collections of Ethereum smart contracts that may serve as benchmarks and tests sets for tool evaluations.

9.1 New Vulnerabilities and Tools

The body of literature in the field of security analysis of smart contracts has been extensive in recent years, with a clearly increasing tendency. The sheer number of publications underpins the fact that Ethereum currently is the *de facto* standard with respect to the research on and the development of smart contracts. Likewise, new smart contract attacks and vulnerabilities keep emerging. With some delay, they are addressed by new or extended tools, with new approaches to vulnerability detection. At the same time, there is not a single tool that covers all documented vulnerabilities. The growing number of tools and related publications substantiates the need for surveys and reviews at regular intervals to guide the interested audience.

9.2 Open Challenges

We identified challenges in all areas discussed in **Section 8**. Regarding *vulnerabilities*, there is still room for improvement in terms of their clear delineation, their arrangement in coherent taxonomies and the classification of their severity. As for the *methods*, the field would benefit from an in-depth discussion on the suitability of detection approaches with respect to specific vulnerabilities. Concerning *tools*, the quality assessment is impeded by a lack of established vulnerability taxonomies and benchmarks. Regarding *benchmark sets*, the ground truth is still sparse, and the available test sets often lack validation in the form of exploits.

9.3 Limitations of Our Review

Threats to the validity of the SLR. The restricted timespan of the study excludes relevant studies recently published, which poses a threat to construct and external validity (Zhou et al., 2016). Data collection and extraction as well as most of the consolidation of the extracted data were carried out in the course of a master's thesis with a tight time budget of six person months. Therefore we did not include a structured snowballing process, which poses threats to both the internal and conclusion validity. This might affect the impartiality of the quality evaluation and the validity of the publication classifications.

Depth. Many naturally arising questions have to remain unanswered in such an endeavor, as they go beyond a pure literature review and would require original research. As an example, the characteristics of a tool, like the vulnerabilities addressed or the methods used, often cannot be determined conclusively from the accompanying publication alone, but would require reading further documentation or the source code. Likewise, assessing the quality of results, the efficiency of detection, the actual practicality, or the maturity of a tool requires additional effort beyond a review. Finally, we did not check the referenced benchmark sets for quality or duplicates.

Platform selection. We deliberately restricted the review to publications on Ethereum smart contracts. We selected this platform for its wealth of documentation, publications, discussions, use cases, handled assets, and overall market value. Some aspects are specific to the predominant programming language Solidity, or to the underlying mechanics of Ethereum and its virtual machine (EVM). However, many aspects of taxonomies, methods and tools are also relevant for other programming languages, virtual machines (like WASM) or even other smart contract platforms influenced by Ethereum.

AUTHOR CONTRIBUTIONS

HR has elaborated the data collection and extraction as well as most of the consolidation as a master thesis supervised by MA who provided critical feedback. MA used parts of the thesis to draft the manuscript. MA and GS revised all sections, especially the methods in automated analysis, and added further consolidations as well as the sections introduction, discussion and conclusion.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fbloc.2022.814977/full#supplementary-material

[1]Etherscan.io.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, & Tools* 2nd Edn. Boston, MA: Pearson Education.

Akca, S., Rajan, A., and Peng, C. (2019). "Solanalyser: A Framework for Analysing and Testing Smart Contracts," in *26th Asia-Pacific Software Engineering Conference (APSEC)* (IEEE), 482–489. APSEC 19. doi:10.1109/APSEC48747.2019.00071

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., and Rubio, A. (2019). "Safevm: A Safety Verifier for Ethereum Smart Contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA: Association for Computing Machinery), 386–389. ISSTA 2019. doi:10.1145/3293882.3338999

Almakhour, M., Sliman, L., Samhat, A. E., and Mellouk, A. (2020). Verification of Smart Contracts: A Survey. *Pervasive Mobile Comput.* 67, 101227. doi:10.1016/j.pmcj.2020.101227

Ante, L. (2021). Smart Contracts on the Blockchain - A Bibliometric Analysis and Review. *Telematics Inform.* 57, 101519. doi:10.1016/j.tele.2020.101519

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). "A Survey of Attacks on Ethereum Smart Contracts (Sok)," in *International Conference on Principles of Security and Trus* (Berlin Heidelberg: Springer), 164–186. vol. 10204 of POST 2017, Lecture Notes in Computer Science (LNCS). doi:10.1007/978-3-662-54455-6_8*t*

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., et al. (2016). "Formal Verification of Smart Contracts," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (New York, NY, USA: Association for Computing Machinery), 91–96. PLAS 16. doi:10.1145/2993600.2993611

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *J. Syst. Softw.* 80, 571–583. doi:10.1016/j.jss.2006.07.009

Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2021). A Survey on Ethereum Systems Security. *ACM Comput. Surv.* 53, 1–43. doi:10.1145/3391195

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., and Chen, T. (2021). "Defectchecker: Automated Smart Contract Defect Detection by Analyzing Evm Bytecode," in *IEEE Transactions on Software Engineering*, 1. doi:10.1109/tse.2021.3054928

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., and Chen, T. (2020). "Defining Smart Contract Defects on Ethereum," in *IEEE Transactions on Software Engineering*, 48, 327–345. doi:10.1109/TSE.2020.2989002

Coblenz, M., Sunshine, J., Aldrich, J., and Myers, B. A. (2019). "Smarter Smart Contract Development Tools," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019* (IEEE), 48–51. doi:10.1109/WETSEB.2019.00013

CORE (2021). CORE Rankings Portal. Available at: https://www.core.edu.au/conference-portal (Accessed August 8, 2021).

Cousot, P., and Cousot, R. (2004). "Basic Concepts of Abstract Interpretation," in *Building the Information Society* (Boston, MA, USA: Springer), 359–366. doi:10.1007/978-1-4020-8157-6_27

di Angelo, M., and Salzer, G. (2019). "A Survey of Tools for Analyzing Ethereum Smart Contracts," in *IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)* (IEEE), 69–78. DAPPCON 2019. doi:10.1109/DAPPCON.2019.00018

Dika, A. (2017). *Ethereum Smart Contracts: Security Vulnerabilities and Security Tools* (Trondheim, Norway: Norwegian University of Science and Technology, Department of Computer Science). Thesis.

Durieux, T., Ferreira, J. F., Abreu, R., and Cruz, P. (2020). "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. (New York, NY, USA: Association for Computing Machinery), 530–541. ICSE 20. doi:10.1145/3377811.3380364

Ferreira, J. F., Cruz, P., Durieux, T., and Abreu, R. (2020a). Smartbugs. Available at: https://github.com/smartbugs/smartbugs (Accessed August4, 2021).doi:10.1145/3324884.3415298

Ferreira, J. F., Cruz, P., Durieux, T., and Abreu, R. (2020b). "SmartBugs," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (New York, NY, USA: Association for Computing Machinery), 1349–1352. ASE 20. doi:10.1145/3324884.3415298

Garfatta, I., Klai, K., Gaaloul, W., and Graiet, M. (2021). "A Survey on Formal Verification for Solidity Smart Contracts," in *Australasian Computer Science Week Multiconference* (New York, NY, USA: Association for Computing Machinery), 1–10. ACSW 21. doi:10.1145/3437378.3437879

Ghaleb, A., and 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* (New York, NY, USA: Association for Computing Machinery), 415–427. ISSTA 2020. doi:10.1145/3395363.3397385

Grishchenko, I., Maffei, M., and Schneidewind, C. (2018). "A Semantic Framework for the Security Analysis of Ethereum Smart Contracts," in *International Conference on Principles of Security and Trust*. Editors L. Bauer and R. Küsters Lecture Notes in Computer Science (Cham: Springer International Publishing) vol 10804, 243–269. doi:10.1007/978-3-319-89722-6_10

Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., and Langendoen, K. (2012). *Modern Compiler Design*. New York, NY, USA: Springer.

Guo, Y.-M., Huang, Z.-L., Guo, J., Guo, X.-R., Li, H., Liu, M.-Y., et al. (2021). A Bibliometric Analysis and Visualization of Blockchain. *Future Generation Comput. Syst.* 116, 316–332. doi:10.1016/j.future.2020.10.023

Gupta, B. C. (2019). *Analysis of Ethereum Smart Contracts - A Security Perspective* (Kanpur: Department of Computer Science and Engineering, Indian Institute of Technology). Master's thesis.

Gupta, B. C., Kumar, N., Handa, A., and Shukla, S. K. (2020). "An Insecurity Study of Ethereum Smart Contracts," in *International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)* Lecture Notes in Computer Science (Cham: Springer) vol. 12586, 188–207. doi:10.1007/978-3-030-66626-2_10

Hartel, P., and van Staalduinen, M. (2019). *Truffle Tests for Free – Replaying Ethereum Smart Contracts for Transparency*. arXiv preprint arXiv:1907.09208.

Hegedűs, P. (2018). "Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts," in *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018* (New York, NY, USA: Association for Computing Machinery), 35–39. WETSEB 18. doi:10.1145/3194113.3194119

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., et al. (2018). "Kevm: A Complete Formal Semantics of the Ethereum Virtual Machine," in *IEEE 31st Computer Security Foundations Symposium* (IEEE), 204–217. CSF 2018. doi:10.1109/CSF.2018.00022

Hu, B., Zhang, Z., Liu, J., Liu, Y., Yin, J., Lu, R., et al. (2021). A Comprehensive Survey on Smart Contract Construction and Execution: Paradigms, Tools, and Systems. *Patterns* 2, 100179. doi:10.1016/j.patter.2020.100179

Jiang, B., Liu, Y., and Chan, W. K. (2018). "Contractfuzzer: Fuzzing Smart Contracts for Vulnerability Detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA: Association for Computing Machinery), 259–269. ASE 2018. doi:10.1145/3238147.3238177

Kalra, S., Goel, S., Dhawan, M., and Sharma, S. (2018). "Zeus: Analyzing Safety of Smart Contracts," in *Network and Distributed Systems Security Symposium* The Internet Society, 1–15. NDSS 2018. doi:10.14722/ndss.2018.23082

Kim, S., and Ryu, S. (2020). "Analysis of Blockchain Smart Contracts: Techniques and Insights," in *IEEE Secure Development (SecDev)* (IEEE), 65–73. SecDev 2020. doi:10.1109/secdev45635.2020.00026

Kitchenham, B., and Charters, S. (2007). "Guidelines for Performing Systematic Literature Reviews in Software Engineering," in *Tech. rep., Software Engineering Group, School of Computer Science and Mathematics* (Keele, England: Keele University).

Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., and Saxena, P. (2019). "Exploiting the Laws of Order in Smart Contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA: Association for Computing Machinery), 363–373. ISSTA 2019. doi:10.1145/3293882.3330560

Leka, E., Selimi, B., and Lamani, L. (2019). "Systematic Literature Review of Blockchain Applications: Smart Contracts," in *International Conference on Information Technologies* (IEEE), 1–3. InfoTech 2019. doi:10.1109/InfoTech.2019.8860872

Liao, J.-W., Tsai, T.-T., He, C.-K., and Tien, C.-W. (2019). "Soliaudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing," in *Sixth International Conference on Internet of Things: Systems, Management and Security* (IEEE), 458–465. IoTSMS 2019. doi:10.1109/iotsms48152.2019.8939256

Liu, H., Liu, C., Zhao, W., Jiang, Y., and Sun, J. (2018). "S-gram: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts," in 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) (New York, NY, USA: Association for Computing Machinery), 814–819. ASE 18. doi:10.1145/3238147.3240728

Liu, J., and Liu, Z. (2019). A Survey on Security Verification of Blockchain Smart Contracts. *IEEE Access* 7, 77894–77904. doi:10.1109/access.2019.2921624

Liu, Y., Li, Y., Lin, S.-W., and Zhao, R. (2020). "Towards Automated Verification of Smart Contract Fairness," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA: Association for Computing Machinery), 666–677. ESEC/FSE 2020. doi:10.1145/3368089.3409740

López Vivar, A., Castedo, A. T., Sandoval Orozco, A. L., and García Villalba, L. J. (2020). An Analysis of Smart Contracts Security Threats Alongside Existing Solutions. *Entropy* 22, 203. doi:10.3390/e22020203

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA: Association for Computing Machinery), 254–269. CCS 16. doi:10.1145/2976749.2978309

Macrinici, D., Cartofeanu, C., and Gao, S. (2018). Smart Contract Applications within Blockchain Technology: A Systematic Mapping Study. *Telematics Inform.* 35, 2337–2354. doi:10.1016/j.tele.2018.10.004

Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A. E. J., and Sharygina, N. (2020). "Accurate Smart Contract Verification through Direct Modelling," in *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Editors T. Margaria and B. Steffen Lecture Notes in Computer Science (LNCS) (Cham: Springer) vol. 12478, 178–194. doi:10.1007/978-3-030-61467-6_12

Mei, X., Ashraf, I., Jiang, B., and Chan, W. K. (2019). "A Fuzz Testing Service for Assuring Smart Contracts," in *IEEE 19th International Conference on Software Quality, Reliability and Security Companion* (IEEE), 544–545. QRS 19. doi:10.1109/QRS-C.2019.00116

[Dataset] MITRE Corp (2006). Common Weakness Enumeration (CWE): A Community-Developed List of Software Weakness Types. Available at: https://cwe.mitre.org (Accessed August 7, 2021).

[Dataset] NCC Group (2018). *Decentralized application security project (DASP) top 10.* Available at: https://dasp.co(Accessed August 8, 2021).

Nguyen, T. D., Pham, L. H., Sun, J., Lin, Y., and Minh, Q. T. (2020). "sFuzz," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (New York, NY, USA: Association for Computing Machinery), 778–788. ICSE 20. doi:10.1145/3377811.3380334

Okoli, C. (2015). A Guide to Conducting a Standalone Systematic Literature Review. *Cais* 37, 43. doi:10.17705/1CAIS.03743

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. (2020). "Verx: Safety Verification of Smart Contracts," in *IEEE Symposium on Security and Privacy (SP)* (IEEE), 1661–1677. SP 2020. doi:10.1109/sp40000.2020.00024

Praitheeshan, P., Pan, L., and Doss, R. (2020a). "Security Evaluation of Smart Contract-Based On-Chain Ethereum Wallets," in *International Conference on Network and System Security*. Editors M. Kutyłowski, J. Zhang, and C. Chen Lecture Notes in Computer Science (LNCS) (Cham: Springer) vol. 12570, 22–41. doi:10.1007/978-3-030-65745-1_2

Praitheeshan, P., Pan, L., Yu, J., Liu, J., and Doss, R. (2020b). *Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey*. arXiv preprint arXiv:1908.08605v3.

Rodler, M., Li, W., Karame, G. O., and Davi, L. (2018). *Sereum: Protecting Existing Smart Contracts against Re-entrancy Attacks*. arXiv preprint arXiv:1812.05934.

Rouhani, S., and Deters, R. (2019). Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access* 7, 50759–50779. doi:10.1109/access.2019.2911031

Samreen, N. F., and Alalfi, M. H. (2020). "A Survey of Security Vulnerabilities in Ethereum Smart Contracts," in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering* (USA: IBM Corp.), 73–82. doi:10.5555/3432601.3432611

Sanchez-Gomez, N., Torres-Valderrama, J., Garcia-Garcia, J. A., Gutierrez, J. J., and Escalona, M. J. (2020). Model-based Software Design and Testing in Blockchain Smart Contracts: A Systematic Literature Review. *IEEE Access* 8, 164556–164569. doi:10.1109/ACCESS.2020.3021502

Schneidewind, C., Grishchenko, I., Scherer, M., and Maffei, M. (2020). "Ethor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA: Association for Computing Machinery), 621–640. CCS 20. doi:10.1145/3372297.3417250

SCImago (2021). *Sjr - Scimago Journal & Country Rank*. Available at: https://www.scimagojr.com/(Accessed August 7, 2021).

Scopus (2021). *Scopus Citescore*. Available at: https://www.scopus.com/sources.uri (Accessed August 7, 2021).

Singh, A., Parizi, R. M., Zhang, Q., Choo, K.-K. R., and Dehghantanha, A. (2020). Blockchain Smart Contracts Formalization: Approaches and Challenges to Address Vulnerabilities. *Comput. Security* 88, 101654. doi:10.1016/j.cose.2019.101654

Snyder, H. (2019). Literature Review as a Research Methodology: An Overview and Guidelines. *J. Business Res.* 104, 333–339. doi:10.1016/j.jbusres.2019.07.039

Soufle (2016). Soufflé: Logic Defined Static Analysis. (Accessed November 1, 2021).

Strong, D. M., Lee, Y. W., and Wang, R. Y. (1997). Data Quality in Context. *Commun. ACM* 40, 103–110. doi:10.1145/253769.253804

SWC Registry (2018). Smart Contract Weakness Classification and Test Cases. Available at: https://swcregistry.io/(Accessed August 8, 2021).

Taylor, P. J., Dargahi, T., Dehghantanha, A., Parizi, R. M., and Choo, K.-K. R. (2020). A Systematic Literature Review of Blockchain Cyber Security. *Digital Commun. Networks* 6, 147–156. doi:10.1016/j.dcan.2019.01.005

Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2020). *A Survey of Smart Contract Formal Specification and Verification*. arXiv preprint arXiv:2008.02712.

Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2022). A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 1–38. doi:10.1145/3464421

Tovanich, N., Heulot, N., Fekete, J.-D., and Isenberg, P. (2021). "Visualization of Blockchain Data: A Systematic Review," in *IEEE Transactions on Visualization and Computer Graphics*, 27, 3135–3152. doi:10.1109/TVCG.2019.2963018

[Dataset] Trail of Bits (2020). (Not So) Smart Contracts. Available at: https://github.com/crytic/not-so-smart-contracts (Accessed August 7, 2021).

Vacca, A., Di Sorbo, A., Visaggio, C. A., and Canfora, G. (2021). A Systematic Literature Review of Blockchain and Smart Contract Development: Techniques, Tools, and Open Challenges. *J. Syst. Softw.* 174, 110891. doi:10.1016/j.jss.2020.110891

Varela-Vaca, Á. J., and Quintero, A. M. R. (2022). Smart Contract Languages. *ACM Comput. Surv.* 54, 1–38. doi:10.1145/3423166

Wang, H., Li, Y., Lin, S.-W., Ma, L., and Liu, Y. (2019a). "Vultron: Catching Vulnerable Smart Contracts once and for All," in *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results* (IEEE), 1–4. ICSE (NIER) 19. doi:10.1109/ICSE-NIER.2019.00009

Wang, H., Liu, Y., Li, Y., Lin, S.-W., Artho, C., Ma, L., et al. (2020). "Oracle-supported Dynamic Exploit Generation for Smart Contracts," in *IEEE Transactions on Dependable and Secure Computing*, 1. doi:10.1109/TDSC.2020.3037332

Wang, S., Zhang, C., and Su, Z. (2019). Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program Lang.* 3, 1–29. doi:10.1145/3360615

Wang, Z., Jin, H., Dai, W., Choo, K.-K. R., and Zou, D. (2020). Ethereum Smart Contract Security Research: Survey and Future Research Opportunities. *Front. Comput. Sci.* 15, 1–18. doi:10.1007/s11704-020-9284-9

Yang, Z., Keung, J., Zhang, M., Xiao, Y., Huang, Y., and Hui, T. (2020). "Smart Contracts Vulnerability Auditing with Multi-Semantics," in *IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)* (IEEE), 892–901. doi:10.1109/compsac48688.2020.0-153

Ye, J., Ma, M., Peng, T., Peng, Y., and Xue, Y. (2019Z). "Towards Automated Generation of Bug Benchmark for Smart Contracts," in *IEEE International*

Conference on Software Testing, Verification and Validation Workshops (IEEE), 184–187. ICST Workshops 2019. doi:10.1109/icstw.2019.00049

Ye, J., Ma, M., Peng, T., and Xue, Y. (2019b). "A Software Analysis Based Vulnerability Detection System for Smart Contracts," in *Integrating Research and Practice in Software Engineering*. Editors S. Jarzabek, A. Poniszewska-Marańda, and L. Madeyski (Cham: Springer), 69–81. vol. 851 of Studies in Computational Intelligence, SCI. doi:10.1007/978-3-030-26574-8_6

Zhang, P., Xiao, F., and Luo, X. (2020). "A Framework and Dataset for Bugs in Ethereum Smart Contracts," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)* (IEEE), 139–150. ICSME 2020. doi:10.1109/icsme46990.2020.00023

Zhang, P., Xiao, F., and Luo, X. (2019). *Soliditycheck: Quickly Detecting Smart Contract Problems through Regular Expressions*. arXiv preprint arXiv:1911.09425.

Zhou, S., Yang, Z., Xiang, J., Cao, Y., Yang, Z., and Zhang, Y. (2020). "An Ever-Evolving Game: Evaluation of Real-World Attacks and Defenses in Ethereum Ecosystem," in *29th USENIX Security Symposium* (USENIX Association), 2793–2810. USENIX Security 2020.

Zhou, X., Jin, Y., Zhang, H., Li, S., and Huang, X. (2016). "A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)* (IEEE), 153–160. doi:10.1109/apsec.2016.031

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.