Assessing smart contracts security technical debts

Ahmadjee, Sabreen; Mera Gomez, Carlos ; Bahsoon, Rami

DOI:
10.1109/TechDebt52882.2021.00010

Document Version
Peer reviewed version

Citation for published version (Harvard):
Ahmadjee, S, Mera Gomez, C & Bahsoon, R 2021, Assessing smart contracts security technical debts. in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt).* vol. 1, Association for Computing Machinery (ACM), pp. 6-15, 2021 IEEE/ACM International Conference on Technical Debt , Madrid , Spain, 22/05/21. https://doi.org/10.1109/TechDebt52882.2021.00010

Assessing Smart Contracts Security Technical Debts

1st Sabreen Ahmadjee
School of Computer Science
University of Birmingham
Birmingham, UK
College of Computer and Information Systems
Umm Al-Qura University
Makkah, Saudi Arabia
smahmadjee@uqu.edu.sa

2nd Carlos Mera-Gómez
ESPOL Polythecnic University
Escuela Superior Politécnica del
Litoral, ESPOL, Facultad de Ingeniería
en Electricidad y Computación
Campus Gustavo Galindo Km 30.5 Vía
Perimetral, P.O. Box 09-01-5863
Guayaquil, Ecuador
cjmera@espol.edu.ec

3rd Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, UK
r.bahsoon@cs.bham.ac.uk

***Abstract*—Smart contracts are self-enforcing agreements that are employed to exchange assets without the approval of trusted third parties. This feature has encouraged various sectors to make use of smart contracts when transacting. Experience shows that many deployed contracts are vulnerable to exploitation due to their poor design, which allows attackers to steal valuable assets from the involved parties. Therefore, an assessment approach that allows developers to recognise the consequences of deploying vulnerable contracts is needed. In this paper, we propose a debt-aware approach for assessing security design vulnerabilities in smart contracts. Our assessment approach involves two main steps: (i) identification of design vulnerabilities using security analysis techniques and (ii) an estimation of the ramifications of the identified vulnerabilities leveraging the technical debt metaphor, its principal and interest. We use examples of vulnerable contracts to demonstrate the applicability of our approach. The results show that our assessment approach increases the visibility of security design issues. It also allows developers to concentrate on resolving smart contract vulnerabilities through technical debt impact analysis and prioritisation. Developers can use our approach to inform the design of more secure contracts and for reducing unintentional debts caused by a lack of awareness of security issues.**

***Index Terms*—smart contract, technical debt, security**

I. Introduction

A smart contract is a decentralised code agreement designed to impose an automatic negotiation of a series of instructions without requiring approval by a central authority [43]. Despite the infancy of smart contracts, their emergence has disrupted several sectors, including cryptocurrencies, financial services, insurance, healthcare, and decentralised management, among others [7]. The widespread usage of this emerging technology has incentivised attackers to exploit its existing security and privacy challenges. Various malicious attacks have been accomplished due to deploying poorly designed or vulnerable smart contracts

Uploading smart contract to the public blockchain is not free as a specific amount of money is required to be paid for each deployment process [46]. Moreover, different from traditional distributed software that can be patched when vulnerabilities are discovered, smart contracts are irreversible and immutable [17]. These properties mean that once a smart contract is deployed in the chain, it cannot be subsequently modified. Therefore, a timely identification of vulnerability is crucial to save business value [11], [12]. In practice, design vulnerabilities do not only originate from 50% of security issues [36] but they are also the most harmful [39] and difficult to identify [25], [45]. Securing smart contracts calls for approaches that can accelerate the identification of root causes of vulnerabilities in the design of such contracts.

The novel contribution of this paper is a debt-aware approach for assessing security design vulnerabilities in smart contracts. The technical debt metaphor has proven to be effective to measure the impact of security weaknesses exploitation in terms of damage to business value [12], the ramifications of negative decisions over time, and locate the design root of security vulnerabilities [25]. Then, we leverage on the metaphor to estimate the monetary cost of redeploying the patched version of the vulnerable contract and the evolution of the debt interest linked to a design vulnerability. Our approach is based on automated analysis tools to discover potential security vulnerabilities caused by design decisions. The evolution of the negative consequences of these security issues in the smart contracts are estimated as an analogy with the concepts of debt principal and interest growth rate.

This research intends to answer the following research questions (RQ): **RQ1:** How to identify design vulnerabilities in smart contracts? What are the specific analysis techniques and tools? **RQ2:** How to quantify the impact of technical debts related to design vulnerabilities in smart contracts?

Although prior works have introduced the idea of technical debt in the context of software security [11], [12], [25], [35], to the best of our knowledge, our work is the first to introduce the metaphor to discuss design vulnerabilities that lead to security issues in smart contracts. Moreover, although previous works have surveyed smart contract vulnerabilities [15], [10], [2], up to our knowledge, this research is the first to focus on design vulnerabilities and map them to their related entries in the Common Weakness Enumeration (CWE) list, which is a catalog of common software and hardware weakness types [20]. Additionally, different from previous efforts that performed empirical evaluations of automated analysis tools

to compare their real capabilities [5], [32], [14], this work characterise a set of automated analysis tools that can discover a group of design vulnerabilities pertinent to smart contracts.

II. PRELIMINARIES

A. Smart Contracts and Ethereum

A smart contract is stored and run on blockchain technology, a distributed database that stores transactions in a decentralised sequence of blocks. Thus, the correct execution of the contract is enforced by the blockchain consensus protocol [43].

Ethereum is the most popular general-purpose blockchain platform that enables developers to deploy smart contracts written using Turing-complete languages such as Solidity. Solidity is an object-oriented language designed specifically for writing blockchain contracts. The programmable contracts are then compiled into bytecode that executes on the Ethereum Virtual Machine (EVM). EVM executes typical cryptocurrency transactions and contracts bytecode, which are a special kind of transaction [44].

In practice, several properties distinguish smart contracts from regular computer programs and make the implications of deploying vulnerable contract more severe. First, the primary distinction is that contracts operate on a decentralised public blockchain network that makes the contract open for inspection, with the state of the contract transparent and traceable by everyone [43]. Second, contracts typically handle monetary transactions that can involve considerable amounts of Ether, an Ethereum cryptocurrency with a current market value equivalent to billions of dollars [2]. The combination of monetary value and public availability makes vulnerable smart contacts compelling targets for attackers. Third, smart contracts are irreversible and immutable. Thus, vulnerable contracts cannot be patched after deployment to the blockchain [17]. The only way to fix the contract, is to deploy a new version with a repair code. However, the old version will remain in the blockchain. Fourth, deploying and executing smart contracts cost an amount of gas (the fuel of computation in Ethereum) [46].

To prevent the abuse of computational resources, in Ethereum, developers and users are required to pay a gas fee to deploy and to execute contracts, respectively [44]. The concept of gas involves the following: (i) **gas cost**, which is a constant amount that determines the computational effort required to execute specific operations; (ii) **gas price**, which denotes the amount of Ether that users are required to pay for each unit of gas; it is dynamic, governed by Ethereum miners, and measured in Gwei (1 Gwei = 0.000000001 Ether); and (iii) **gas fee**, which is the incentive received by miners in exchange for the computional resources used to execute the operations of a contract and the building of blocks; gas fee is converted to Ether and paid to miners.

B. The Common Weakness Scoring System

The Common Weakness Scoring System (CWSS™) [21] provides a quantitative mechanism to score unfixed CWEs found in software. The system supports a prioritisation of weaknesses in terms of their potential negative consequences. The scoring formula is based on three metrics: *Base Finding*, which consists of several factors that estimate the level of technical impact once the weakness is successfully exploited, the accuracy of finding, and the effectiveness of the existing control; *Attack Surface*, whose factors estimate how easy the attacker could exploit the weakness; *Environmental*, which involves factors that refer to a specific operational context such as the potential impact to the business and the likelihood of discovery. Each metric factor is assigned a numeric value that is calculated based on specific formulas, and the sub-score of each metric is multiplied with each other to generate the final CWSS score in a range between 0 and 100.

C. Technical Debt

Technical debt is a metaphor devised to capture how the value of software engineering decisions evolves over time [19]. Specifically, the metaphor supports the identification of the roots of a sub-optimal decision, the estimation of its value, and the monitoring of the environmental trade-offs in which the decision was made [13]. The metaphor also supports attributes that are intrinsic to debts in finance such as principal and interest [41]. In the metaphor, the principal represents the value of the gap between the ideal and the actual decision, whereas the interest represents the additional effort that needs to be incurred to pay back the principal.

Technical debt has been applied in architectural design to value the gap between the ideal and an actual decision, to identify the architectural root of an issue, and to manage the debt incurred by a decision [41], among others. Since some roots of security issues tend to be intertwined with architectural decisions, the metaphor has also been used to identify security vulnerabilities [25], prioritise the attention to security weaknesses [12], and to estimate the consequences on business value if vulnerabilities are exploited [11].

III. OUR APPROACH FOR ASSESSING TECHNICAL DEBTS

This section presents our approach to assessing the security technical debts incurred in smart contracts design. The assessment involves detecting design issues and quantifying their consequences to security if remain unfixed.

We have defined the steps that support the creation of secure by design smart contracts:

1) Identify security design vulnerabilities in a smart contract.
   a) Run automated security analysis tools on the smart contract source code to obtain a list of potential vulnerabilities.
   b) Perform a manual analysis complementing the automated analysis to uncover any missed vulnerability.
   c) Determine the design vulnerabilities and map them to related weaknesses to highlight the root cause of the issues.
   d) Classify the design vulnerabilities per design flaw categories to determine the negative technical impact upon the contract.

2) Measure the ramifications of the identified design vulnerabilities.
   a) Estimate the monetary cost of technical debt principal by calculating the gas fee for redeploying a patched version of the vulnerable contract.
   b) For each vulnerability identified, estimate technical debt interest value by quantifying the negative security consequences and the interest growth rate over time.

Our approach is a debt-aware assessment approach, as it facilitates visualisation of the debt incurred by exploitable flaws created while designing smart contracts. Furthermore, applying this approach, developers can be aware of the long-term consequences of security design issues. Subsequently, the developer can prioritise the debt based on both the monetary cost and the interest value related with vulnerability violations.

A. Identification of Security Design Vulnerabilities

Building our assessment approach involves two steps: aggregating vulnerabilities caused by flaws in smart contracts design, Subsection III-A1, and selecting security analysis tools that assists the identification process, Subsection III-A2.

1) Mapping Design vulnerabilities to Security Design Weaknesses: Security flaws coming from design decisions have been reported as the main cause of software security problems [25]. If these flaws remain unaddressed, these can be viewed as root cause of technical debt, with consequences observed through accumulation of interests over time [11] [25]. Additionally, if security software engineers are aware of these issues but they do not fix them, these workarounds can be considered as (self-) admitted technical debts.

Although several efforts, from industry and academia, illustrate smart contract vulnerabilities, they have missed to distinguish between coding and architectural design vulnerabilities. Therefore, in this study, we present a set of vulnerabilities rooted in the architectural design of smart contracts and their related security weaknesses. The aggregated set is classified based on the security impact of such issues.

Security weaknesses are flaws in software that may lead to exploitable security vulnerabilities. Therefore, the identification of weaknesses can assist us in understanding the security problems and performing root cause analysis. Our study leverages Common Weakness Enumeration (CWE™) [20], which is a community-developed list of common security weaknesses that may appear in architecture, design, or implementation of software. Thus, we aim to map vulnerabilities in contract design to security weaknesses in the CWE catalog that stands out regarding adoption and scope of coverage. Although CWE does not refer to any weaknesses particular to smart contracts, it depicts associated weaknesses at higher abstraction layers.

We followed a systematic procedure to collect and map each design vulnerability in contract design architecture to the corresponding weaknesses in CWE. This procedure consisted of two steps.

First, we collected a comprehensive list of security vulnerabilities from academic papers that surveyed existent vulnerabilities in smart contracts [10] [15] [2] [1]; we also considered Ethereum community [29], wiki [8], and developers' blogs [47] [34] that listed and explained exploitable flaws not discussed in the literature. Specifically, we extracted information about the vulnerabilities, such as their descriptions, ways of exploitation, and preventive techniques that can be used to avoid them. The collected information assisted in knowing the negative impacts of these issues on the contracts and the cost of fixing them. In addition, the information also helped to determine the vulnerabilities that result from the flaws manifested at the contract's design stages. Second, we analysed each collected vulnerability to map it with the subset of weaknesses (438/1248) of CWE. This subset represents weaknesses that can be introduced during a design stage.

To reduce inherently biases in the mapping process, two authors separately worked over all the collected vulnerabilities. After completing the analysis, results were compared and double-checked with Smart Contract Weakness Classification (SWC) [38] Registry. This registry established by a group of developers, auditors, and researchers at ConsenSys Diligence [3] that provides smart contracts developers with a system analog to CWE. However, at the time of writing, only 20 architectural design weaknesses are listed in the registry. Finally, each disagreement was discussed with a third author.

We observed that the identified design vulnerabilities can be classified based on their impact into ten categories. Front-Running, Time Manipulation, Denial of Services (DoS), Broken Access Control, Arithmetic Issues, and Bad Randomness. These first six categories are also presented in Decentralised Application Security Project (DASP) [9] taxonomy. Other categories are Sensitive Data Exposure and Using Components with Known Vulnerabilities, which are the third and ninth category, respectively in Open Web Application Security Project (OWSAP)-Top 10 Security Risks [31]. The two remaining categories are Improper Inheritance and Modularity Violation, which are the flaws that violate object-oriented design principles. Since Solidity is an objected oriented language, the contracts written by this language are also prone to these types of design flaws.

Table I describes each design flaws category, whereas Table II shows an example of mapping the design vulnerabilities classified under DoS to relevant CWEs. Our replication package, described in Subsection III-C carries full details of the mapping.

Our classification allows the designers to select a specific design flaws category and visualise all related vulnerabilities and weaknesses. It may also serve as a guide for architects to avoid common security architectural issues when creating smart contracts.

2) Selection of Security Analysis Tools: The analysis of potential vulnerabilities in smart contracts needs to be performed before its deployment to the immutable environment of the blockchain, in which refactoring or updating the contract is costly and not trivial.

Recently, a growing number of security analysis tools have emerged to detect common vulnerabilities and bad practices in Ethereum smart contracts written in Solidity. Although these

TABLE I
DESCRIPTION OF DESIGN FLAWS CATEGORIES

Category	Source of Category	Description
Front-Running	DASP	A type of race condition where a malicious user can steal the solution and submit a transaction with a higher gas price to make their transaction assigned to the block before the victim.
Time Manipulation	DASP	The timestamp of the block is adjusted by malicious miners to their own advantage.
Denial of Services	DASP	The attacker can make the contract inoperable temporarily or permanently.
Arithmetic Issues	DASP	An arithmetic operation that reaches the max or min size of a type and presents incorrect results that compromise contract security and reliability.
Bad Randomness	DASP	The random number generator is written in a way that is predictable and exploitable.
Sensitive Data Exposure	OWSAP-10	The developer does not adequately protect critical information related to the contract and assumes that private type variables cannot be read.
Using Components with Known Vulnerabilities	OWSAP-10	Malicious or deficient components, such as libraries or off-chain data sources, where the developer unaware of all the running code.
Improper Inheritance	Object Oriented Design Flaws	When a contract inherits another contract, the presence of multiple variables with the same name in both contracts might lead to unintended effects.
Modularity Violation	Object Oriented Design Flaws	This consists of a tight coupling between contracts that makes them frequently change together.

tools have been developed by different teams such as academic teams [42], community teams [17], and industrial teams [22], most of the existing tools are inaccurate in finding the security issues as they yield a considerable number of false positives [5] [32]. Furthermore, each tool only detects a limited scope of vulnerabilities. Consequently, the combination of several existing tools is essential to increase coverage and accuracy of vulnerabilities detection.

In this study, we selected a set of security analysis tools that help in identifying design vulnerabilities in smart contracts. To do that, we investigated the academic literature, searched the Internet, and scanned Github. The tools selected for our study were those matching the following criteria (C):

C#1. The tool is free, publicly available and specific information about it can be found in at least one official source.

C#2. The tool is up to date.

C#3. The tool can take source code of the contract as an input.

C#4. The tool identified at least one design vulnerability or bad practice related to design.

After exploring all the collected tools, we identified only 9 tools that met the criteria. These tools are: Slither [6], SmartCheck [40], and Securify [42] which perform static analysis techniques. Mythril [23] which is a symbolic analysis tool and Manticore [22] which perform dynamic symbolic execution identification. Another tool is sFuzz [24] which applies a fuzzing testing technique. Solhint [33] and Ethlint [4] perform static analysis to identify security issues and bad practices. Final tool is Mythos which applies a combination of dynamic symbolic execution and fuzzing techniques.

TABLE II
A VULNERABILITIES MAPPING FOR DOS TO CWE CATEGORISATION

Design Vulnerabilities	CWE	Design Weaknesses
Exception handling problem	CWE-703	Improper Check or Handling of Exceptional Conditions
Non-validated arguments	CWE-20	Improper Input Validation
DoS by external contract/Call	CWE-703	Improper Check or Handling of Exceptional Conditions
Calculates the upper bond of Gas	CWE-400	Uncontrolled Resource Consumption
Costly pattern/Costly loop	CWE-400	Uncontrolled Resource Consumption
Reachable SELFDESTRUCT	CWE-28	Improper Access Control

B. Measurement of Negative Consequences of Design vulnerability in Smart Contracts

Besides detecting vulnerabilities, it is also crucial to estimate the associated implications of taking a shortcut by not fixing those issues. To this end, we adopt the notions of principal and interest to quantify the debt cost and value related to each identified vulnerability in a smart contract. Quantifying the cost and value of the security debts aids developers to apply a cost-effective technique and justifies the investment in fixing security problems.

1) Estimation of Gas Cost (Quantifying the Principal): Immutability in smart contracts restricts developers' ability to patch vulnerabilities after deploying the contract to the blockchain. The developer needs to upload a new version of the contract after fixing those vulnerabilities, and this requires additional gas consumption with the associated expenses. As a result, the need to refactor vulnerable deployed contract has financial implications.

Our aim is to calculate the gas consumption fee for redeploying a contract to estimate technical debt principal (P). This estimation provides useful insights for developers regarding the cost of repairing vulnerabilities in the deployed contracts. We use the following formula:

$$P_{SecurityDebt} = Gas\_D(c) + Gas\_U(c) \quad (1)$$

where Gas_D indicates the cost of gas required to deploy the repaired contract (c), and Gas_U indicates the cost of the gas required to update the contract (c) for a given issue.

Gas Cost Required to Redeploy Repaired Contract. The cost of the gas required for deployment depends on the size of the smart contract. The number of functions (NoF) and lines of

code (LoC) are both influencing factors, and the more complex the contract is, the more gas consumption is required. The uploading cost fee is computed using *gas_price* × *gas_cost*, where the former is the value of a unit of gas as specified by the market, and the latter is mostly determined by the summation of the following factors:
G_{create}, 32000 gas, paid for a CREATE operation,
$G_{transaction}$, 21000 gas, paid for every transaction,
$G_{codedeposit}$, 200 × |o| gas, paid per byte for a create operation,
|o| amount of bytecode in the compiled contract,
Execution costs, gas paid for necessary computation processes.

The last factor refers to the costs associated with the part of the code that requires to be executed before the creation of the contract, such as initialising the state variables whose values are permanently stored in the contract storage, and executing a constructor. If the constructor requires a lot of computation to generate the bytecode, then there will be extra expense. Appendix G in the Ethereum yellow paper [44] shows the costs, in gas, of several opcode operations.

Gas Cost of Update Pattern. Since redeployment results in a new contract with a new address, an update pattern needs to be utilised to avoid using the vulnerable deprecated contract. The developer can use a self-destruct opcode, which costs $Gselfdestruct = 5000$ gas, to destroy the contract. Another option is to upload the main contract with a proxy smart contract, which has a changeable variable that stores the main contract's address. Thus, once an updated contract version is released, the value of this version is updated.

2) Estimation of Security Consequences (Quantifying the Interest): In the security context, the interest value represents the undesirable effects that can result if the vulnerability is exploited. The longer the exploitable design flaw remains unaddressed in the deployed contract, the higher the chance the debt interest associated with this flaw grows. Accordingly, three factors are considered when estimating the accumulated interest: CWSS score, contract activity level, and contract lifespan.

CWSS Score. We adopted the CWSS to estimate the severity of identified weaknesses in a contract. The CWSS framework provides different scoring methods; in this study, we use the targeted method, which assesses individual design weakness. Multiple factors are used to quantify CWSS scores in smart contracts, such as mapping the vulnerabilities to the related CWEs and the proposed design flaws categories. This information allows the negative impact on the contract in the case of an attack to be estimated. The estimated score generated by CWSS allows for technical debt items to be visualised and those debts that lead to more severe consequences to be addressed. In practice, addressing contract security design issues early, in the pre-deployment stage, minimises debt.

Accumulated interest. Debt interest increases when exploited contracts become extremely common like, for example, the hacked Parity Multi-Sig Wallet contract [2]. This was a smart contract for a multiple signature wallet that had a critical, exploitable vulnerability, which allowed an attacker to steal millions of dollars. The debt interest also increases if the exploitation of vulnerabilities in the contract has irreparable consequences, as is the case with suicidal vulnerability. With this vulnerability, any arbitrary account can kill a contract, causing it to stop functioning and locking its ether [42]. The accumulation of interest associated with vulnerabilities in smart contracts is difficult to quantify; however, we find that the activity level and the lifespan of the contract are useful for estimating interest growth.

Contract Activity Level (CAL). CAL refers to the expected number of active users and the number of transactions that a contract is expected to deal with. This factor can be predicted from statistical information about contracts provided on websites, such as State of the ĐApps [27], which is a website with a curated list of smart contract-based applications. This website ranks the contracts and categorises them based on their activity level. Statistical research [28] shows that the top three high-activity contract categories in 2020 are games, currency exchanges, and gambling.

Contract Lifespan (CLS). CLS refers to the time, measured in days, between the contract's deployment and its last execution. Most smart contracts live between 2 and 800 days [16], and the average age of smart contracts is 295.6 days [28]. A contract can be short-lived, medium-lived, or long-lived [28], [16]. A contract is long-lived when it is expected to be executed for a long interval or even for as long as the network exists. Conversely, a short-lived contract is mostly designed to be executed for a limited time and then either the interaction with it ends or it is destroyed. This type of contract typically includes few operations, involves few fixed users, and has limited transactions. Thus, the security implications of breaching this contract are not significant compared to a long-lived, highly active contract in which there is interaction with thousands of customers and thousands of transactions are accepted over a long period of time.

Therefore, during the CLS period (i.e., before the contract reaches its maturity stage), the accumulated interest (AI) of the security technical debt can be estimated as follows:

$$AI_{SecurityDebt} = CWSSscore \times CAL \times CLS \quad (2)$$

Nevertheless, we acknowledge that some issues may go beyond, and the interest can be adjusted accordingly. For this paper, we focus on interest within the CLS period, as this enables action to be taken before contract redemption to avoid the accumulation of the issues that are rooted in technical debt.

The developer can assign a point scale to each contract activity level category and lifespan category. The multiplication of the scores of the three factors determines the accumulated interest.

C. Replication Package for Replicability

The data source and replication package for our experiments are publicly available [1] to assist in verifiability and reproducibility of our results.The package includes: (i) the complete mapping of all aggregated design vulnerabilities and their rated CWE entries; (ii) the list of all founded analysis tools; (iii) the

[1]https://bitbucket.org/Smart_Contract/assessing-smart-contracts-security

TABLE III
EXPERIMENT DATASET. FOR EACH VULNERABLE CONTRACT, WE PROVIDE DESIGN FLAWS CATEGORIES OF ITS FLAWS (DFC), NUMBER OF DESIGN VULNERABILITIES (VULNS), AND LINES OF CODE (LOC).

Contracts	DFC	#Vulns	#LOC
FindThisHash	Front-Running	1	9
EtherLotto	Time Manipulation / Bad Randomness	1	20
Roulette	Time Manipulation	1	14
Lottopollo	Time Manipulation / Bad Randomness	1	24
DosAuction	DoS	1	13
SimpleToken	DoS / Access Control Broken	1	65
Etheraffle	DoS/ Bad Randomness	2	122
DosNumber	DoS	1	28
AccessControl	Access Control Broken	1	53
BlockdBuildDemo	Access Control Broken	3	62
FunctionTypes	Access Control Broken	2	18
OddEven	Sensitive Data Exposure	1	22
Transaction_malleablity	Sensitive Data Exposure	1	77
Token	Arithmetic Issues	1	17
TokenSaleChallenge	Arithmetic Issues	1	20
CEOThrone	Improper Inheritance	1	21

set of vulnerabilities claimed to be identified by each tool; (iv) details of inclusion criteria that were not met by each of the excluded tools; (v) the dataset used in the experiment; and (vi) the detailed results of each step of the approach.

IV. EXPERIMENTATION

To demonstrate our approach, the following example shows how the steps of our approach are performed to assess the potential security-related debts in smart contracts.

A. Experiment Setup

We collected a dataset of 16 representative vulnerable smart contracts that are either actual contracts identified as vulnerable or explicitly programmed to demonstrate a specific vulnerability. This dataset comes from several publicly-available resources: (i) *GitHub* repositories such as SWC Registry, not-so-smart-contracts [26], and Solidity-Security [18]; (ii) *Ethernet* [30], an online game for smart contract attacking challenges; and (iii) *blog.positive* [37], which publishes articles discussing vulnerabilities in smart contracts.

We intentionally chose a dataset of known vulnerable contracts to examine the ability of the 9 selected security analysis tools to identify vulnerabilities. Our selection of smart contracts enables a discussion of their source codes and permits the publication of our results for replication without users being involved in legal and privacy issues.

All the tools are installed and run on the same computer utilising Solidity compiler- solc (v 0.7.1), under Ubuntu 18.04.5 operating system. Only sFuzz provides a web-based interface for using the tool, then there was no need to install anything related to it.

B. Experimental Study

Figure 1 represents the overall procedure of our experiment.

Step 1.a. We first ran the automatic security tools against each contract in our dataset to identify the design vulnerabilities. As mentioned in the previous section, different types of tools were selected to enhance the detection abilities of design vulnerabilities. We started the operation process by running the tools that perform static analysis techniques. After the static analysis has been completed, operating the tools that perform dynamic analysis techniques was the next step for obtaining deeper results. We finalised the analysis by running the tool's applied fuzzing techniques.

Step 1.b. We conducted a manual analysis to complement the previous step as the 9 tools could not detect all the known vulnerabilities in the dataset. We annotated the known vulnerabilities and detected several other flaws that were not mentioned in the online source with regard to the vulnerable contracts. We ended up with a set of possible vulnerabilities that was composed of flaws not related to the design.

Step 1.c.&1.d. We checked the aggregated set of design vulnerabilities and the aligned CWEs to only record the issues that belonged to that set. Each contract had one or more vulnerabilities mapped in at least one of the design flaw categories. There were 20 distinct vulnerabilities in our dataset. Table III shows each collected contract, its name, the categories of its flaws, the number of design vulnerabilities, and the lines of code.

Step 2.a. The cost fees required to deploy the patches contracts were calculated by Formula 1 to quantify the technical debt principal. A self-destruct updated pattern were considered when calculating the total gas cost. At the time of experimenting, the average gas price was 126 Gwei, and Ether's price was around $500 US.

Step 2.b. We estimated the security risk severity scores related to identified vulnerabilities and weaknesses using the CWSS formula that explained in Subsection II-B . We used the OWSAP method to estimate the technical and business impact factors, in case vulnerabilities are exploited. This method suggests dividing the technical impact into sub-factors aligned with the traditional security areas of concern: loss of confidentiality, integrity, availability, and accountability. Additionally, our proposed categories of design flaws support determining the technical impact as each category shows the ramifications of the exploitation. Similarly, the method divides the business impact factor into sub-factors common to many businesses: financial damage, reputation damage, non-compliance, and privacy violation. Each sub-factor has several scored options. We selected one of the options and then averaged the scores for each factor, technical and business impacts, to decide their severity level.

There are other required factors to calculate the CWSS score, such as required privileges; authentication and access vector to perform the attack; and the acquired privilege after accomplish it. These factors were determined based on the information collected about each vulnerability and the way of exploiting it. The knowledge about the vulnerable contracts and security analysis performed assisted in determining factors such as finding confidence, likelihood of discovery and exploit.

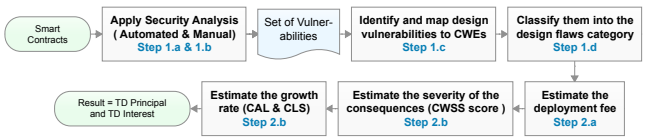

Fig. 1. Experiment execution steps

In addition to the CWSS score, the contract's activity level and lifespan factors have to be estimated to visualise the potential accumulated interest. A six-point scale is used to determine the activity level, where 6 refers to the top 3 highly active contract's category, and 1 refers to the lowest 3 categories. The ranking of the categories is informed by the State of the DApps website. Since most smart contracts live between 2 to 800 days, we divided the lifespan into three intervals and gave each interval a score as follows: (i) Short-lived, 1-266 days, 0.17 score; (ii) Medium-lived, 267-533 days, 0.35 score; (iii) Long-lived, 534-800+ days, 0.5 score. The accumulated interest for each unfixed vulnerability were calculated by Formula 2. The value of the final score is between 0 and 300. Noticeably, we set those particular scores to visualise how the interest could redouble. However, contract developer can customise the scores based on their context.

V. Results and Discussion

In this section, the collected dataset of some represented vulnerable contracts are analysed and discussed with respect to our research questions. Additionally, we discuss the practical implications of our findings and we outline the potential threats to their validity.

A. Identification of Design Vulnerabilities (RQ1)

Locating design vulnerabilities in smart contracts source code is the main step in our assessment approach. Figure 2 presents the results of executing 9 security analysis tools on 16 vulnerable contracts in order to locate their design vulnerabilities. It shows the percentage of vulnerabilities that each tool was able to detect per design flaws category.

Nine tools, in combination, were only able to locate 70% (14/20) of all the vulnerabilities. Most of the tools were generally able to detect a high percentage of vulnerabilities classified in the categories *Time Manipulation*, *Improper Inheritance*, and *Bad Randomness*. However, the tools under performed when it came to identifying vulnerabilities of the categories *Front-Running* and *Sensitive Data Exposure*. The 9 tools failed to detect four vulnerabilities linked to *Access Control Broken*, one vulnerability linked to *DoS*, and another linked to *Sensitive Data Exposure*. Throughout the analysis, we observed that the output of some tools was noisy, as they failed to disregard the false warnings and detected unreal flaws.

Figure 2 indicates that the tools demonstrate different ability to locate design vulnerabilities. The tool Mythril surpassed all other tools as it was able to identify more vulnerabilities in the 16 contracts than any of the others. Securify is the only tool that detected the vulnerability, Transaction Ordering Dependency, linked to the *Front-Running category*. Although the documentation of several tools such as Manticore claimed that they could detect this vulnerability, they failed to identify it. Similarly, Smartcheck is the only tool that identified the vulnerability, Unencrypted Private Data On-Chain, in the *Sensitive Data Exposure* category. However, the output of Smartcheck does not reveal a clear explanation of the identified flaws. Mythril and Mythos both provide informative output as they link the identified vulnerabilities to related SWC. Securify resulted in the most confusing output among all the tools because it showed a large number of false alarms.

Our study emphasises that conducting manual analysis to complement automated ones is essential to discover most vulnerabilities. This is because, as our analysis experiment shows, the scope of issues addressed by state-of-the-art tools is limited. Additionally, the accuracy of their analysis needs to be increased by reducing the likelihood of false-positive alarms that confuse the developers and discourage them from leveraging the results. Even though an empirical study [5] claimed that combining different types of analysis tools yields more vulnerability coverage, the existing tools are not powerful enough to replace manual analysis. Thus, taking a shortcut by deploying the contract to the public without performing both automated and manual inspection processes might lead to invisible exploitable flaws that are prone to attacks.

Through the analysis, all the detected flaws were found in our aggregated set of design vulnerabilities and their associated weaknesses. In contrast, some of these detected flaws, including no restricted write and no restricted transfer, did not exist in the SWC Registry which provides a set of classified issues that come up in smart contract development. Moreover, our proposed design flaw categories cover all the detected flaws, while the current DASP taxonomy is not extensive enough to include all types of security design issues that affect smart contracts. Unlike DASP, our classification includes vulnerabilities of the following categories: Sensitive Data Exposure, Using Components with Known Vulnerabilities, Improper Inheritance, and Modularity Violation.

Classifying vulnerabilities based on their impact and mapping them to a wider group of agreed-upon weaknesses (CWEs) come with several benefits: (i) it provides a common language for defining security architectural design issues in smart contracts; (ii) it offers a simple way to classify security issues in such contracts; and (iii) it generates useful insights into the root-causes of vulnerabilities, and the negative impacts posed by the exploitable ones.

B. Estimation of Negative Consequences (RQ2)

The second step in our approach is to estimate the ramification of deploying vulnerable contracts to the public blockchain in terms of debt principal and accumulated interest. Figure 3 shows debt principal and interest of 10 vulnerable contracts from our dataset out of 16. The figure represents the estimated monetary cost of refactoring each contract and the value

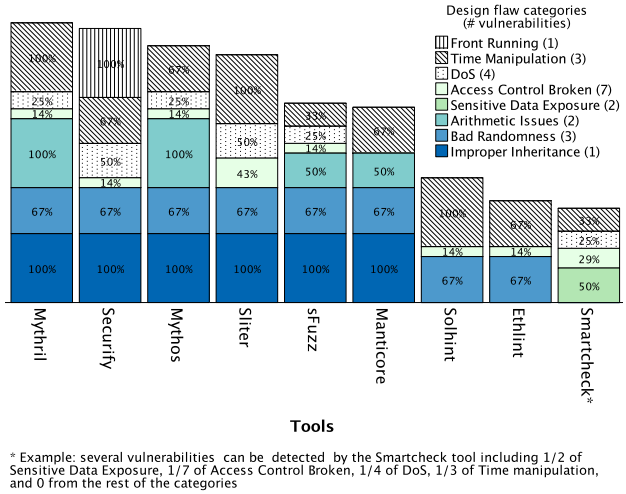

Fig. 2. Identified vulnerabilities per category by each tool.

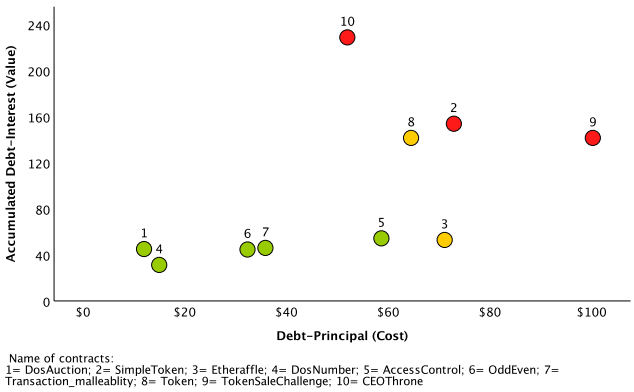

Fig. 3. Principal and interest of ten vulnerable contracts.

of the accumulated interest in relation to the effects of the vulnerabilities over time, if they are left unfixed.

For the sake of visualising and discussing the data, we have categorised the coordinates of the (cost, value) points in the graph into three categories: high, medium, and low, depending on their severity. Given the score for values in this range [0,300], we defined (100<value<=200) as medium, values greater than 200 as high, and values lower than or equal 100 as low. As in our example the highest cost is $100.25, we defined ($33.33<cost<=$66.66) as medium, a cost greater than $66.66 as high, and a cost lower than or equal to $33.33 as low. The final degrees of severity are then derived from both the cost and the value using Table IV. As such, vulnerabilities in contracts 2, 9, and 10 fall into the high category; vulnerabilities in contracts 3 and 8 into the medium category; and those in contracts 1, 4, 5, 6, and 7 into the low category.

Estimating the cost and value related to security violations in smart contracts facilitates decision-making regarding which issues need more attention when seeking to reduce technical debt. For instance, a vulnerability in the CEOThrone contract is assumed to be in the high category. The cost of gas required to fix the issue that has not been fixed before deployment is 897,200, which is equivalent to $51 at the time of writing. The size of this contract is relatively small compared to other contracts in our dataset. There is only one operation in the constructor, and there are no state variables that need to be initialised.

However, the contract suffers from variable shadowing vulnerability, which occurs when a variable declared within a child contract has the same name as a variable declared in a parent contract. In the case of CEOThrone, this allows an unauthorised node to withdraw the balance of the contract, leading to a significant technical and business impact. The likelihood of discovering and exploiting this issue is also high, since most of the analysis tools were able to detect it. The vulnerability in this contract allows any attacker to gain the required privilege to exploit it. CEOThrone is a contract created for a game and it can thus be assumed that the contract will be highly active and have a long lifespan. As such, we estimated the accumulated security debt interest for this contract at around 228.9, which is a high score compared with other contracts.

Beyond the results of our experiments, our assessment approach can significantly reduce the cost of developing smart contracts, in particular in cases where a contract is big and, as such, has a potentially higher cost, or in the case of fixing an issue in a contract that requires modifying and redeploying other dependent contracts. We provided a quantitative way of indicating the relative costs and values of publicly deploying vulnerable contracts. Developers can thus make informative decisions before uploading a contract to the blockchain. Additionally, they can prioritise the resolution of issues based on the available quantitative information. Overall, the assessment approach presented here supports reducing the introduction of unintentional debt caused by unawareness of security issues rooted in smart contract design while increasing the visibility of such issues and helping manage the debt more strategically.

C. Threats to Validity

A potential threat to internal validity is related to the possibility of considering alternative security flaws scoring methods. However, unlike other scoring systems, the CWSS can be applied at the early stages of the development process. CWSS provides support if there is incomplete information, as most of its factors have values for uncertainty and flexibility, such as Unknown and Not applicable. Furthermore, CWSS facilitates a thorough estimation of the security consequences of uploading vulnerable contracts. It considers 16 factors compared to the only four factors in the risk rating of the OWSAP method.

Another threat relates to the potential subjectivity of CWSS results. We mitigated this possible risk as follows: (1) we proposed categories of design flaws that enable precise determination of technical impact; (2) conducting the security analysis creates awareness of the ease and likelihood of discovering any flaws; (3) in the proposed estimation Formula 2, the CWSS score is accompanied with smart contract activity level and lifespan factors.

TABLE IV
OVERALL SEVERITY LEVEL

	Value		
Cost	**High**	**Medium**	**Low**
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

A potential threat to external validity relates to the fact that the contracts we have considered in our experiment are relatively small and simple (approximately 1.5KB) - typical size of commonly used contracts in practice as contracts are recommended to be simple and not complex. Nevertheless, the same steps and analysis can scale up to larger ones, often not exceeding 24KB in practice. Referring to our 10 out of 16 selected contracts in our study, the estimated monetary costs required to pay for repairing them were not significant. However, the quantitative approach provided for calculating the debt principal is applicable to any size and type of smart contract. As mentioned earlier, the monetary cost is dynamic as it depends on the gas price and the Ether price, either of which may significantly increase. For instance, in January 2018 the price of Ether rose dramatically to $1066 US. Hence, uploading even a small and simple contract might cost a significant amount of money.

A potential threat to construct validity is the completeness of the aggregated set of design vulnerabilities. We mitigated this risk by conducting a thorough inspection of the academic papers, the Ethereum community, Wiki pages, and developers' blogs. Nevertheless, the set of vulnerabilities supplied is continually evolving because of the high potential for the emergence of new exploitable security design flaws in smart contracts.

VI. RELATED WORK

We discuss closely related work, covering smart contract vulnerabilities; empirical evaluations of automated analysis tools that are relevant to our work; and security technical debt.

Smart Contract Vulnerabilities. Previous studies have illustrated, discussed, or surveyed various security vulnerabilities in blockchain and smart contracts. Li et al. [15] reviewed the security vulnerabilities in blockchain, the corresponding attacks, and suggested several security solutions, without differentiating between Bitcoin and Ethereum. Similarly, the authors in [10], classified the security vulnerabilities based on blockchain generation. They also provided a detailed explanation of known vulnerabilities and subsequent potential attacks. Unlike these studies, we focus on design vulnerabilities in smart contracts run on Ethereum. Other studies such as that of Atzei et al. [1], discussed 12 known vulnerabilities in Ethereum smart contracts, and classified them into three categories based on the level where they presented: the Solidity level, the EVM level, and the blockchain level. Similarly, authors in [2] provided the same classification of vulnerabilities in Ethereum smart contracts; however, they provided a more comprehensive list. In contrast, we aggregated vulnerabilities caused by flaws in a contract's architectural design, mapped them to their related CWE entries, and classified them based on their impact.

Automated Analysis Tools. As several automated tools have recently emerged to analyse the security of smart contracts, empirical studies have been conducted to compare their real capabilities and the techniques used. Durieux et al. [5] carried out a systematic evaluation of several state-of-the-art automated tools and discussed their accuracy and efficiency. We also followed a systematic method when it came to selecting the 9 tools that were used in the vulnerabilities identification process. But different from other studies, we identified and analysed a set of tools able to discover a subset of design vulnerabilities. Parizi et al. [32] performed an assessment of four static analysis tools with regard to 10 vulnerable smart contracts. In contrast, in our analysis, besides using static analysis tools, we included tools that use dynamic or fuzzing techniques. Additionally, we analysed their ability to identify design vulnerabilities on 16 vulnerable contracts. Leid et al. [14] compared the effectiveness of symbolic and fuzzy testing tools by evaluating their effectiveness when analysing smart contracts. Despite a large number of available automated analysis tools, they only considered three tools in their assessment.

Security Technical Debt. The nature of work presented in this paper is generally related to applying technical debt to raising the visibility of security risks, and the costly consequences of deploying vulnerable smart contracts. Despite the vast contributions on technical debt in software, only few studies have looked at security-related debts including [11] [12] [25] and [35]. In particular, Izurieta et al. [11] established an approach to prioritise security technical debt in a software system. This was related to CWEs entries by leveraging the CWSS scoring systems. Unlike their approach, our study proposes a mechanism to estimate the accumulated debt interest, not only by using the CWSS score, but also by including the activity level and lifespan of the smart contract under consideration. Additionally, we quantify the debt principle of refactoring the vulnerable contract. In [12], authors mapped attack tactics to the related posed consequences of the exploitable vulnerabilities. This helps to prioritise vulnerabilities that require the most attention to reduce technical debt. In [25], the authors applied a preliminary experiment to show the correlation between technical debt and software vulnerabilities. A recent study [35], regarding security debt, discussed the concept of managing security risk by leveraging technical debt. They emphasised that the combination of software security engineering techniques and technical debt increases the security of software systems. Our work goes beyond existing work on security technical debt and is the first to explicate security technical debts in smart contracts.

VII. CONCLUSION

In this article, we have presented a debt-aware approach for assessing security design vulnerabilities in smart contracts. We used nine state-of-the-art tools to widen the detection

abilities of security design issues in smart contracts. We use CWE catalogue when analysing the identified vulnerabilities and their weaknesses. We adapted the community-informed scoring mechanism to consider contract activity level and the contract lifespan. The combination helps security software engineers to estimate the accumulated debt interests related to design vulnerabilities in a contract. Debt principal was quantified by calculating the gas fee required to redeploy the patched version of a vulnerable contract. Experiment results demonstrated that our approach can allow developers to visualise and prioritise technical debts, rooted in unaddressed smart contract design vulnerabilities. The approach can increase the visibility of debts and their ramifications.

In future work, we seek to automate the estimation steps of our approach to support faster and more efficient analysis. Automated support will ensure consistency in the analysis and allow security software engineers to quickly estimate the implications of applying alternative security options when designing smart contracts.

REFERENCES

[1] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptol. ePrint Arch.*, 2016:1007, 2016.

[2] H. Chen, M. Pendleton, L. Njilla, and S. Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[3] CONSENSYS. mythx. https://mythx.io/.

[4] R. Dua. Protofire. https://github.com/duaraghav8/Ethlint.

[5] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541, Seoul South Korea, 2020. ACM/IEEE.

[6] J. Feist, G. Grieco, and A. Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, Montreal, QC, Canada, Canada, 2019. ACM/IEEE.

[7] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría. Blockchain and smart contracts for insurance: Is the technology mature enough? *Future Internet*, 10(2):20, 2018.

[8] github wiki. List of security vulnerabilities, Apr. 2019.

[9] N. Group. Decentralized application security project. https://dasp.co/.

[10] H. Hasanova, U.-j. Baek, M.-g. Shin, K. Cho, and M.-S. Kim. A survey on blockchain cybersecurity vulnerabilities and possible countermeasures. *International Journal of Network Management*, 29(2), 2019.

[11] C. Izurieta, K. Kimball, D. Rice, and T. Valentien. A position study to investigate technical debt associated with security weaknesses. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 138–142. IEEE, 2018.

[12] C. Izurieta and M. Prouty. Leveraging secdevops to tackle the technical debt associated with cybersecurity attack tactics. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019.

[13] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21, 2012.

[14] A. Leid, B. van der Merwe, and W. Visser. Testing ethereum smart contracts: A comparison of symbolic analysis and fuzz testing tools. In *Conference of the South African Institute of Computer Scientists and Information Technologists 2020*, pages 35–43, 2020.

[15] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107, 2020.

[16] M. Lohr and S. Peldszus. Maintenance of long-living smart contracts. In *Software Engineering (Workshops)*, Innsbruck, Österreich, 2020. ceurws.

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, Vienna Austria, 2016. ACM.

[18] A. Manning. solidity-security-blog. https://bit.ly/2Oi2xsa.

[19] C. Mera-Gómez, F. Ramírez, R. Bahsoon, and R. Buyya. A multi-agent elasticity management based on multi-tenant debt exchanges. In *Proceedings of the 12th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2018)*. IEEE, 2018.

[20] MITRE. Common weakness enumeration. https://cwe.mitre.org/.

[21] MITRE. Common weakness scoring system. https://cwe.mitre.org/cwss/.

[22] M. Mossberg. manticore. https://github.com/trailofbits/manticore.

[23] B. Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 9:54, 2018.

[24] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh. Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 778–788, New York, NY, USA, 2020. Association for Computing Machinery.

[25] R. L. Nord, I. Ozkaya, E. J. Schwartz, F. Shull, and R. Kazman. Can knowledge of technical debt help identify software vulnerabilities? In *9th Workshop on Cyber Security Experimentation and Test ({CSET} 16)*, 2016.

[26] T. of Bits. (not so) smart contracts. https://bit.ly/2ZX2VyK.

[27] S. of the DApps. stateofthedapps. https://www.stateofthedapps.com/.

[28] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, pages 1–41, 2020.

[29] f. openzeppelin. Discuss about smart contract security patterns, vulnerabilities, hacks and best practices, Nov. 2020.

[30] openzeppelin team. ethernaut. https://ethernaut.openzeppelin.com/.

[31] OWASP. Open web application security project. https://bit.ly/30aNtzc.

[32] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, page 103–113, USA, 2018. IBM Corp.

[33] Protofire. solhint. https://github.com/protofire/solhint.

[34] Y. Riady. Common smart contract vulnerabilities and how to mitigate them, 2018.

[35] K. Rindell, K. Bernsmed, and M. G. Jaatun. Managing security in software: Or: How i learned to stop worrying and manage the security technical debt. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–8, 2019.

[36] J. C. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223. IEEE, 2017.

[37] I. Security. blog.positive. https://blog.positive.com/.

[38] SmartContractSecurity. Swc registry. https://cwe.mitre.org/cwss/.

[39] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, et al. Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[40] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, Gothenburg Sweden, 2018. ACM/IEEE.

[41] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[42] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *18 Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.

[43] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang. Blockchain-enabled smart contracts: architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019.

[44] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[45] U. Yavo. Design vulnerabilities: They hide and you can't catch them. https://bit.ly/37S4Bhl.

[46] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury. Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–32, 2020.

[47] K. Zipfel. New smart contract weakness: Hash collisions with multiple variable length arguments, Jan. 2019.