This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

Self-Admitted Technical Debt in Ethereum Smart Contracts: A Large-Scale Exploratory Study

Amir Mohammad Ebrahimi, Gustavo A. Oliva, and Ahmed E. Hassan

Abstract—Programmable blockchain platforms such as Ethereum offer unique benefits to application development, including a decentralized infrastructure, tamper-proof transactions, and auditability. These benefits enable new types of applications that can bring competitive advantage to several business segments. Nonetheless, the pressure of time-to-market combined with relatively immature development technologies (e.g., the Solidity programming language), lack of high-quality training resources, and an unclear roadmap for Ethereum creates a context that favors the introduction of technical debt (e.g., code hacks, workarounds, and suboptimal implementations) into application code. In this paper, we study self-admitted technical debt (SATD) in smart contracts. SATD refers to technical debt that is explicitly acknowledged in the source code by developers via code comments. We extract 726k real-world contracts from Ethereum and apply both quantitative and qualitative methods in order to (i) determine SATD prevalence, (ii) understand the relationship between code cloning and SATD prevalence, and (iii) uncover the different categories of SATD. Our findings reveal that, while SATD is not a widespread phenomenon (1.5% of real-world contracts contain SATD), SATD does occur in extremely relevant contracts (e.g., multi-million contracts). We also observed a strong connection between SATD prevalence and code cloning activities, leading us to conclude that the former cannot be reliably studied without taking the latter into consideration. Finally, we produced a taxonomy for SATD that consists of 6 major and 26 minor categories. We note that several minor categories are bound to the domain of blockchain and smart contracts, including gas-inefficient implementations and Solidity-induced workarounds. Based on our results, we derive a set of practical recommendations for contract developers and introduce open research questions to guide future research on the topic.

Index Terms—Self-Admitted Technical Debt, SATD, Maintenance, Smart Contracts, Ethereum, Blockchain

1 Introduction

Blockchain technology enables digital transactions to be executed in a completely decentralized and secure manner. Due to these characteristics, blockchain is seen as a technology that has the potential to transform the way businesses are conceived and operate. In a recent book [1], Gartner noted that blockchain can generate up to US$3.1 trillion in new business value by 2030.

A key player in the blockchain landscape is the *Ethereum platform*. Ethereum brought a significant innovation when it first launched back in 2015: the support for *smart contracts*. Smart contracts are computer programs that serve as the building blocks for the development of blockchain-powered applications [2]. These blockchain-powered applications are often referred to as *decentralized applications* (DApps). Blockchain platforms that support smart contracts and DApps are often called *programmable blockchains*.

Ethereum is the most popular programmable blockchain. As of April 08th 2022, 73.6% of all DApps listed on the *State of the DApps* website (a curated directory of DApps) are deployed on Ethereum. Moreover, 41 out of the top-50 ranked DApps are deployed on Ethereum. Recurrent DApp categories include NFTs (digital collectibles) and DeFi (decentralized finance). Such a popularity highlights Ethereum's potential to create competitive advantage in several business markets and segments.

Nevertheless, there is the other side of the coin. Ethereum's significant popularity also gives rise to disproportionate business expectations in companies. Indeed, NFTs have been placed at the *Peak of Inflated Expectation* in the 2021 edition of the famous Gartner's Hype Cycle [3]. The overall hype around blockchain technology (including the "fear of missing out") has also been documented several times in the literature [2, 4, 5].

The aforementioned context creates pressure for early (and possibly premature) adoption of Ethereum as a platform for the development of DApps. In this paper, we conjecture that such a pressure might have translated into a hasty, rushed out development of smart contracts. The fact that the most popular programming language for smart contracts (Solidity) is also substantially younger compared to more traditional languages (e.g., C, Java, and Python) likely aggravates the problem. More specifically, the language's technical ecosystem (e.g., static analyzers, compilers, debuggers, and IDEs) is considerably less mature than those of traditional languages [6]. Another natural consequence of Solidity's young age is the limited number of experts in that language, possibly reducing the availability of high quality training material (e.g., online tutorials) and knowledge sharing.

To evaluate our aforementioned conjecture, we set out to investigate whether smart contracts contain *technical debt* (TD). TD is a metaphor coined by Cunningham [7], which refers to the cost of additional work caused by choosing a quick and sub-optimal solution instead of a better, more robust one that would take longer to implement. Similarly to recent TD studies in SE [8–10], we focus on *self-admitted technical debt* (SATD). SATD refers to TD that is explicitly acknowledged in the source code by developers via code comments [11]. We collect a raw dataset with over 5.7 million smart contracts, extract real-world contracts from it, and study the prevalence and types of SATD. More specifically, we investigate the following research questions:

RQ1) How prevalent is SATD in smart contracts? We employed the Jitterbug tool [12] to detect SATD in relevant (real world, non-toy) smart contracts. Next, we analyze the activity level

- *Amir Mohammad Ebrahimi, Gustavo A. Oliva, and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, Canada.*
  E-mail: {amir.ebrahimi,gustavo,ahmed}@cs.queensu.ca

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

(number of received transactions) and currency influx (total inbound USD) of contracts with SATD. We observe that:

SATD is not a widespread phenomenon, as only 1.5% of all relevant contracts actually contain SATD. Nonetheless, some contracts with SATD are remarkably relevant in practice: some have received millions of transactions and others have received ETH amounts worth millions of USD. In particular, contracts with SATD typically received more transactions than those without SATD.

RQ2) How does SATD prevalence in smart contracts relate to code cloning? Recent studies have indicated that smart contracts are cloned with much higher rates than those observed in traditional software [13–16]. This makes code cloning a strong confounding factor in any analysis of smart contracts. In this RQ, we investigate SATD prevalence and its evolution in light of code cloning. We observe that:

There exists a strong connection between SATD prevalence and code cloning. Indeed, over 93% of SATD contracts are clones. The majority of cloned contracts are systematically deployed over time with Contract Accounts (CAs) having a bold role in their creation in a programmatic manner. In addition, the vast majority of textually unique SATD contracts has a single debt regardless of their size. Developers mostly employ a TODO keyword in a simple single-line comment ("//") to self-admit technical debt.

RQ3) What types of SATD do smart contracts contain? In this RQ, we wish to have a deeper understanding of the types of sub-optimal solutions that developers implement and acknowledge. Such a deeper understanding should provide insights into techniques and tools to better assist contract developers. To this end, we manually analyzed a statistically representative sample of SATD instances (comments) using open coding and carefully crafted a taxonomy of the different types of SATD. We observed that:

We found six types of SATD including: code (38.7%), requirement (26.5%), test (6.62%), design (5.6%), documentation (1.4%) and forward compatibility (1%) self-admitted technical debt. We also provide a detailed view via a taxonomy that specialize our SATD types into 26 low-level categories. Many of such categories are specific to the smart contracts domain. We document each category along with real-world examples taken from our studied contracts.

To the best of our knowledge, this is the first in-depth, large-scale study of SATD in smart contracts. Our key contributions are as follows: (i) describing what SATD means in the special context of smart contracts (Section 2), (ii) designing a methodology for empirically detecting and studying SATD in smart contracts, (iii) characterizing the prevalence of SATD in real-world smart contracts and associating that prevalence to code cloning activities in Ethereum (RQ1 and RQ2), (iv) designing a taxonomy for SATD (RQ3), and (v) laying out a discussion around open research challenges and practical implications of our results in an effort to bootstrap future research in the topic (Section 5). From a broader perspective, this paper raises awareness about the inherent challenges of addressing technical debt in the context of smart contracts. A supplementary package is available online[1].

Paper structure. Section 2 discusses how SATD matters to smart contracts and particularly the impacts of contracts immutability on SATD payment. Section 3 describes our data collection approach. Section 4 shows our motivation, approach, and findings for each of our research questions. Section 5 discusses the implications of our findings while also highlighting opportunities for future research. Section 6 discusses related work. Section 7 lists the threats to the validity of our study. Finally, Section 8 concludes our study. Since this paper touches on several blockchain concepts and their intricacies, we include a background section in Appendix A.

1. https://bit.ly/3Lfc4uY. Once the paper is accepted, this supplementary package will be moved to GitHub

2 Background and Motivation

2.1 What does technical debt mean for smart contracts?

To answer this question, we refer to the classical 4+1 architecture view model proposed by Kruchten [17]. The two views that matter to us are the *development view* (a.k.a., implementation view) and the *physical view* (a.k.a., deployment view):

Development view. This view illustrates a system from a programmer's perspective, focusing on software management. Identifying, tracking, and solving technical debt is a key aspect of software management [7]. **From a development view, managing technical debt in smart contracts is not any different than managing technical debt in traditional software systems.** Indeed, smart contracts are simply source code files that need to be managed just like any other piece of source code (e.g., Java code). More specifically, smart contracts implement the business logic of a DApp (decentralized application), which is the name given to applications that run on the blockchain (e.g., Ethereum). Just like any software-intensive application, DApps are commonly developed using a repository on a coding platform (e.g., GitHub). That repository will thus contain the smart contracts used by that DApp. What do these contracts look like? In Solidity, the most popular smart contract programming language, developers typically create a new .sol file for each smart contract of the DApp (this is analogous to how developers typically create a .java file for each class of an object-oriented Java application). Most importantly, these smart contracts will undergo changes (commits) to implement the ever-changing business rules of a real-world application. Contracts may change due to non-functional requirements, such as the need to be more efficient (consume less gas). Due to time-to-market pressure, urgency to release, immaturity of software development tools [18], and a fast-evolving ecosystem, DApp developers might add technical debt to smart contracts. For example, assume that a smart contract contains a suboptimal implementation of a function, which results in this contract burning more gas than it should. Releasing a DApp with this contract means that those sending transactions to this DApp (e.g., end-users) will have to pay more money than ideal, rendering the DApp more expensive overall.

Physical view. The physical view refers to the topology of software components on the physical layer (hardware), as well as the physical connections between these components. The physical view is thus directly associated with how applications are to be deployed and upgraded. This is where things deviate substantially in the blockchain world compared to traditional software systems. As a baseline, let us consider a server-side application developed with web services. The deployment scheme of this type of application has evolved drastically over the years. In particular, online upgrades can nowadays be done seamlessly, with zero down-time, and in a variety of ways (e.g., blue/green deployment and canary deployment [19]). Online upgrades are enabled by a capability known as *traffic rerouting*. Before hitting

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

the actual web service providing the functionality, the requests are first intercepted by some server (e.g., Nginx[2]). This server can then be configured to redirect users to the new application version. For instance, with canary deployment, the new version of the application is deployed alongside the old one. The systems engineer then controls who gets to use the new version. This is done by monitoring the process and gradually expanding the percentage of users who have access to the new version. The key point that we want to emphasize is that, **in traditional server-side applications, traffic rerouting is typically decoupled from the application logic (i.e., some component outside of the application is in charge of it)**. In public blockchains platforms such as Ethereum, upgrades happen in a very different way. DApp developers have no control over Ethereum, meaning they cannot deploy a server like Nginx to redirect traffic. Developers also cannot hack a deployed smart contract, since (i) only the bytecode is stored in the blockchain and (ii) that bytecode is immutable (i.e., cannot be changed) by design of the blockchain. **Traffic rerouting thus has to be handled by the application itself (e.g., via proxy contracts [20]**. Therefore, unless the DApp was designed to be upgradable, upgrades can only be manually performed. Manual upgrades are costly and difficult to perform. For instance, we note that contracts are stateful and store data. Hence, data migration plans needed to be crafted and executed. In particular, some DApps have an architecture in which a new contract is deployed for each new user of the application. In that case, a DApp upgrade will involve dealing with the state of not one, but several contracts. More generally, there is no established governance protocol for manual DApp upgrades involving user migration. In summary, **upgrading a DApp is typically more complicated compared to web service applications.** What does this mean in the scope of our paper? It depends on the stakeholder:

- Contract authors. They have to carefully plan whether their contracts should contain technical debts, since removing it is not as trivial. This involves, for instance, (i) assessing its impact to the end-users of the DApp (e.g., when a gas suboptimal function is implemented) and (ii) assessing the difficulty to debug problems (e.g., when a hacky, less-readable solution is employed). In addition, deploying a new repaired contract (i.e., a contract without SATD) will always cost money. As with any transaction, the cost is given by: `units of burnt gas * price of gas * value of ETH`. For decentralized applications in which a contract is deployed for each end-user, this cost can become potentially very large (especially if the contract's source code is large) [21, 22].

- Contract users. Users include both DApp end-users and client DApps (that might integrate with the DApp in question). These users will likely experience the effects of contracts with technical debt for a longer time compared to users of web service applications.

- Contract cloners. Prior literature indicated the existence of a high number of type-1 contract clones (i.e., identical copies) in Ethereum [15, 16, 23]. Those who clone contracts with SATD are subject to the same observations that we made for contract authors.

Smart contract developers thus need to be careful when adding technical debt to a smart contract, since releasing application upgrades is less straightforward compared to traditional server-side applications. In the cases where technical debt is (expected to be) unavoidable, then DApps should likely be designed from the get-go to support contract upgrades.

Given the unique role of technical debt in smart contracts, in this first study we aim to characterize the occurrence of (self-admitted) technical debt in real-world smart contracts. In particular, we aim to understand if real-world contracts contain SATD (RQ1), whether the prevalence of SATD is associated with the high cloning activity happening in Ethereum (RQ2), and what types of SATDs there are (RQ3).

2.2 Illustrative examples of SATD in smart contracts

Example 1: Missing documentation. Listing 1 depicts a contract with an external function (i.e., a function exposed in the contract's API) that lacks documentation. The SATD is indicated in the form of a `TODO`. The contract in question implements a multi-signature (a.k.a. multisig) wallet[3]. In simple terms, a multi-signature wallet contract implements a cryptocurrency wallet that requires the confirmation of multiple people (i.e., multiple signatories) before the funds can be transferred from/to the wallet.

The contract in question was developed by an Ethereum co-founder (Gavin Wood) and was the most popular multi-signature contract of its time. It was the backbone of the Mist DApp, a wallet application developed by the Ethereum Foundation. Mist was eventually deprecated and end-users had to manually interact with the contract (i.e., send transactions to the contract) in order to withdraw cryptocurrency or move it to another wallet. The lack of documentation for the API only made things even harder for end-users[4].

Given the non-triviality of the contract's source code (1311 lines of code, 51 functions), lack of documentation can lead to misunderstandings about how this contract operates. Since this contract directly manipulates funds, such misunderstandings can lead to financial problems in practice. If some code is directly managing some form of currency with a clear market value, then the behavior of such code should be very well documented. Indeed, as a pull-request on GitHub states[5]: "*As this multisig can and is used in practice for very large funds, I thought it could be a good idea to enhance its documentation to better reflect how to use it and some common pitfalls.*"

```
1  [...]
2  contract multisig {
3    [...]
4    // EVENTS
5    [...]
6    // FUNCTIONS
7    // TODO: document
8    function changeOwner(address _from, address _to) external;
9    function execute(address
           _to, uint _value, bytes _data) external returns (bytes32);
10   function confirm(bytes32 _h) returns (bool);
11 }
12 [...]
```

Listing 1: Missing source code documentation.

Example 2: Workaround for a Solidity limitation. Our second example refers to a contract's function[6] that implements a workaround to retrieve the return value of a call to a built-in Solidity function known as `ecrecover`. The workaround

2. http://www.nginx.com

3. Contract Address: 0xebacd5907d96b80e7d0f94af0292e4e50742d813

4. See https://ethereum.stackexchange.com/questions/112727 and https://support.mycrypto.com/how-to/sending/how-to-interact-with-a-multisig-contract

5. https://github.com/ethereum/dapp-bin/pull/73

6. Contract Address: 0xc78bb6c8848abe10c3a398fa1801926215d27f6f

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

is implemented using *inline assembly*, which is a low-level instruction set from Solidity.

```
/*
The following function has been written by Alex Beregszaszi (@axic),
use it under the terms of the MIT license Duplicate Solidity's
ecrecover, but catching the CALL return value
*/
function safer_ecrecover
      (bytes32 _hash, uint8 _v, bytes32 _r, bytes32 _s)
       internal returns (bool _success, address _recoveredAddress) {
  /*
  We do our own memory management here. Solidity uses memory offset
  0x40 to store the current end of memory. We write past it (as
  writes are memory extensions), but don't update the offset so
  Solidity will reuse it. The memory used here is only needed for
  this context.
  FIXME: inline assembly can't access return values
  */
  bool ret;
  address addr;
  assembly {
      let size := mload(0x40)
      mstore(size, _hash)
      mstore(add(size, 32), _v)
      mstore(add(size, 64), _r)
      mstore(add(size, 96), _s)
      ret := call
            (3000, 1, 0, size, 128, size, 32) // NOTE: we can reuse
            the request memory because we deal with the return code.
      addr := mload(size)
  }
  return (ret, addr);
}
```

Listing 2: Workaround for a Solidity limitation.

3 Data Collection

Our study entails collecting the source code of smart contracts and detecting SATD in it. Figure 1 provides an overview of our data collection process.

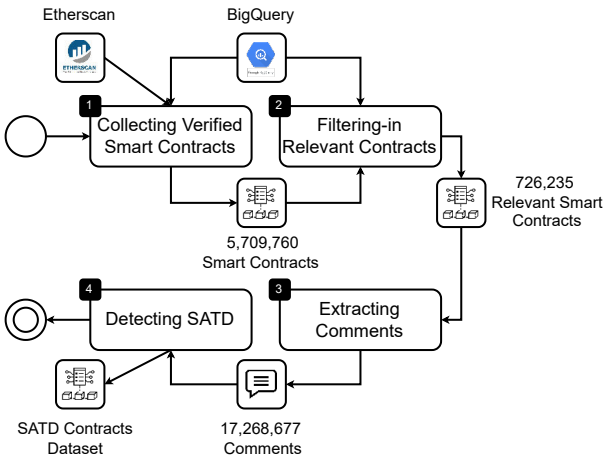

Fig. 1: An overview of our data collection process.

Step-1: Collecting smart contracts As the input of the process, we require a dataset of smart contracts. The dataset used in this work is a collection of all verified Solidity contracts hosted on Etherscan as of March 18th, 2019, which encompasses over 5.7 million contracts (5,709,760 to be exact). To collect verified contracts from Etherscan, we used the Ethereum dataset[7] from the Google BigQuery platform and obtained the addresses of *all* deployed contracts by means of an SQL query. Next, we used Etherscan's REST API to retrieve the contracts' source code. More specifically, for each address, we first checked whether the contract's source code is available and written in Solidity. In the positive case, we retrieved the source code (otherwise we skipped it). We focus exclusively on the Solidity language because it has historically accounted for over 99% of all verified contracts in Ethereum[8].

Step-2: Filtering-in relevant contracts. We use a heuristic to filter-in real-world, non-toy contracts (henceforth referred to as *relevant contracts*). We define a smart contract as relevant if it satisfies either of the following criteria:

- Activity level: the contract received at least C_1 transactions
- Currency influx: the contract received at least C_2 USD worth of ETH.

To set C_1, we first determined the number of successful inbound transactions for each of the 41,769,738 contracts that had been deployed to Ethereum as of May 25th, 2021. We use this date to increase the likelihood of contracts to be seen as relevant, especially those created close to March 2019 (see Step 1). To do so, we once again leveraged the Ethereum dataset from Google BigQuery. Next, we made C_1 correspond to the 99th percentile of the distribution, which resulted in $C_1 = 29$ (i.e., 1% of all contracts in Ethereum received 29 or more transactions).

As for the second criterion, we set $C_2 = 100$ USD as the minimum deposit that a contract must have received (currency influx) to be considered relevant. Our rationale is that contract users would refrain from spending 100 USD just to play or experiment with a toy contract. To compute the USD currency influx of a contract, we first determined a daily ETH-to-USD exchange rate by averaging ETH's opening and closing prices from August 7th, 2015 (date of first contract transaction on Ethereum) to May 25th, 2021. The daily opening and closing prices where directly obtained from the Yahoo Finance[9] website. Finally, we used the Ethereum dataset from Google BigQuery once more to select transactions that sent ETH to contracts. Finally, we converted the ETH amount to USD using the aforementioned exchange rates.

Having applied our criteria to all contracts, we identified 726,235 (12.72%) relevant contracts. More specifically, 123,259 (16.97% of all relevant contracts, 2.16% of all contracts) and 649,607 (89.44% of all relevant contracts, 11.38% of all contracts) are included via the first and second criterion respectively. Only 46,631 (6.41% of all relevant contracts, 0.82% of all contracts) are included by satisfying both criteria. Such a result highlights the complementary nature of the two criteria.

Step-3: Extracting comments. Once we obtained our set of relevant contracts, we then extracted their source code comments. We implemented an ANTLR parser using a popular Solidity grammar specification[10]. A total of 17,268,677 comments were extracted from our relevant contracts.

Step-4: Detecting SATD. We analyzed and compared the existing SATD detection approaches [11, 12, 24–28] to choose an appropriate one for our study. Our criteria include the year and venue where the tool was published, the type of approach (machine learning, NLP, pattern-matching), whether it is automated, the types of debt that the tool detects, and whether a replication package is available. After careful consideration, we chose the Jitterbug tool [12] and employed the functionality

7. Dataset name: `bigquery_public_data.crypto_ethereum`

8. https://etherscan.io/dashboards/contract-statistics
9. https://finance.yahoo.com/quote/ETH-USD/history?p=ETH-USD
10. https://github.com/antlr/grammars-v4/blob/master/solidity/Solidity.g4

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

associated with the detection of the "easy-to-find SATD". A more detailed evaluation of the candidate SATD detectors is described in Appendix B.1.

To use Jitterbug's "easy-to-find SATD" detector, we trained its pattern recognition algorithm on a curated corpus of SATD comments made available by the tool's authors [12]. More information about this dataset is available in Appendix B.2. After training the algorithm, the output was a set of four SATD indicator patterns, namely: `todo`, `fixme`, `hack`, and `workaround`.

Finally, we looked up the four indicator patterns in our Solidity comments dataset and counted every match as one SATD instance. For conciseness purposes, we refer to contracts that contain at least one SATD instance as *SATD contracts*. Analogously, we refer to contracts without SATD as *non-SATD contracts*.

Summary

- Smart contracts collection period: August 2015 to March 2019 (43 months).
- Data sources: Etherscan, Google Big Query
- Pieces of collected data: Source code of smart contracts, metadata associated with smart contracts (as of May 25th, 2021)
- Number of studied smart contracts: 5,709,760
- Self-admitted technical debt (SATD) detector: Jitterbug (easy-to-find SATD) [12].

4 Findings

This section describes the motivation, approach, and findings associated with each of our research questions.

4.1 RQ1: How prevalent is SATD in smart contracts?

Motivation. Technical debt is detrimental to software quality [7]. Yet, paying off debt is particularly challenging in the context of smart contracts (Section 2). Hence, as a starting point, in this RQ we investigate the prevalence of SATD in smart contracts.

Approach. We first determine the proportion of contracts with SATD. As described in Section 3, we employed Jitterbug's *easy-to-find* component to detect SATD in *relevant* smart contracts. Next, we characterize SATD contracts by analyzing their activity level and currency influx. We compare data distributions using parametric statistical tests and the Cliff's Delta (δ) measure of effect size. We use the following thresholds to interpret d [29]: *negligible* for $|\delta| \leq 0.147$, *small* for $0.147 < |\delta| \leq 0.33$, *medium* for $0.33 < |\delta| \leq 0.474$, and *large* otherwise.

Findings. Observation 1) ***1.5% of relevant contracts contain SATD.*** We detected 10,549 relevant contracts with SATD, which accounts for 1.5% of all relevant contracts (726,235). Hence, SATD is not a widespread phenomenon. Nonetheless, this result also indicates that relevant (i.e., real-world, non-toy) contracts are being put in production in spite of them having technical debt.

Observation 2) ***SATD occurs in extremely relevant smart contracts.*** The left-hand side of Figure 2 depicts the activity level of SATD contracts. The curve is considerably right-skewed, with an acute peak in the 100 to 1k range. We observe a long tail after the 95% percentile (red dotted line). Such a long tail indicates that there exists a small set of contracts that is extremely relevant in practice (from an activity level perspective). For reference, the top-5 most active contracts received from 5.6M to 33.6M transactions. They are: FiatTokenProxy (a proxy contract used in the context of the USD Coin, which is a cryptocurrency that is pegged to USD), KittyCore (the core contract of the CryptoKitties game, which is arguably the most popular Ethereum NFT-based game), CardMigration (one of the core components of Gods Unchained game, which is another popular decentralized trading card game), GasToken2 (a popular productivity contract that enables users to store gas when it is cheap and use it later on when it becomes expensive), and AdminUpgradeabilityProxy (a contract that combines upgradeability features with an authorization mechanism for administrative tasks).

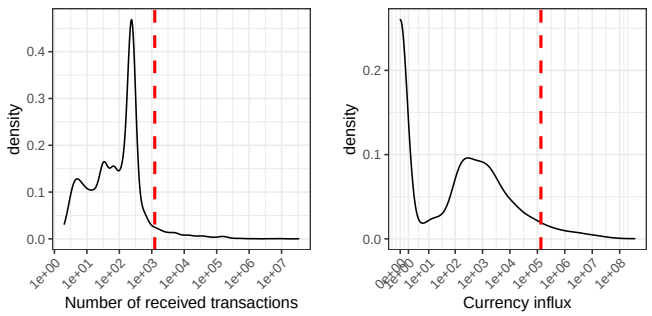

Fig. 2: The distributions of activity level (left-hand side) and currency influx (right-hand side). The x-axes are log-transformed. Red lines denote the 95th percentile.

The right-hand side of Figure 2 depicts the currency influx of SATD contracts. Similarly to the activity level curve, the currency influx curve also has a long tail after the 95th percentile (albeit shorter). We thus conclude that there exists an additional small set of contracts that are also extremely relevant in practice (from a currency influx perspective). For reference, the top-5 richer contracts received from 84.3M to 352.2M USD worth of Ether. They are: Wyvern Exchange (A decentralized exchange for trading any kind of non-fungible assets which is utilized by OpenSea, the largest NFT marketplace), the next top three contracts are three copies of the MultiSig Wallet contract, and Etheroll (a smart contract for placing bets on users' provably-fair dice game using Ether).

Observation 3) ***SATD contracts have a higher activity level than those without SATD.*** We observe that the median number of received transactions is 118 for SATD contracts, which is roughly 17 times higher than that for non-SATD contracts. A two-tailed Mann-Whitney test (α = 0.05) indicates that the difference between the distributions is indeed statistically significant (p-value $\leq$ 0.05). Calculation of Cliff's Delta indicates that the magnitude of the difference is large (δ = 0.6). Hence, SATD contracts are used more frequently than those without SATD. This observation highlights once more the practical relevance of SATD contracts.

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

Summary

RQ1: How prevalent is SATD in smart contracts?

SATD occurs in ~1.5% of all relevant contracts. That is, some contracts are being put in production in spite of them having technical debt. In particular:

- SATD occurs in contracts that have received millions of transactions and/or millions of USD.
- SATD occurs in contracts associated with popular DApps, such as CryptoKitties (most popular NFT game) and OpenSea (largest NFT marketplace).
- Contracts with SATD typically received more transactions than those without SATD.

4.2 RQ2: How does SATD prevalence in smart contracts relate to code cloning?

Motivation. Prior research has shown that cloning is prevalent in Ethereum [13–16]. In particular, Kondo et al. [15] reported that more than 79.2% of all of their studied contracts had at least one duplicate. Therefore, we conjecture that SATD prevalence is associated with code cloning.

As a preliminary study, we analyzed the proportion of duplicate SATD instances. Our rationale is that it is unlikely that independent developers would write the exact same code comment in their contracts. We discovered that 11,115 (95.3%) out of the 11,710 SATD instances are duplicates, which is a very strong sign of cloning. Hence, in this RQ, we further examine the relationship between SATD and code cloning.

Approach. To investigate the relationship between SATD and code cloning, we performed the following analyses:

- *On the prevalence of cloning among contracts with SATD.* As an initial step, we determine the number of contract clones, clone clusters, and clones per cluster among SATD contracts. To obtain those numbers, we employ the approach proposed by Kondo et al. [15]. First, we use the *Cloc*[11] tool to strip blank lines and comments out of our collection of SATD contracts. Next, we apply the *Deckard* [30] clone detector. We chose Deckard because it is one of the few well-tested clone detection tools that supports the Solidity language. Since Deckard operates on Abstract Syntax Trees (ASTs), we need to ensure that it can correctly parse all the preprocessed contracts. After investigating Deckard's log files, we identified 210 (2%) contracts that Deckard failed to parse. We excluded these incompatible contracts from this analysis. As for Deckard's configuration, we set its similarity threshold to 0.79 based on the sensitivity analysis performed by Kondo et al. [15] and leave other settings at their default values. In terms of output, Deckard detects clone pairs at the *code fragment level* (line ranges). Since our goal is to determine whether contracts with SATDs are clones of each other, we post-process these fragment level clone pairs to obtain file-level clones. We refer to a set of file-level clones as a *clone cluster*. Thus, we consider a contract to be a *clone* if its cluster is *not* a singleton (i.e., the size of its cluster is higher than or equal to 2).
- *On the evolution and deployment authorship of cloning.* To have a grasp on how cloning numbers evolve and who deploys clones (deployment authorship), we study the top-10 largest clone clusters. To study cloning evolution, we sort contracts within each clone cluster in ascending order of deployment timestamp. Next, we analyze these deployment timestamps to uncover trends in the evolution of cloning activities. With regards to deployment authorship, our goal is to determine who is behind the cloning activities associated with each cluster. To this end, we study the percentage of clones created by externally-owned accounts (EOA) versus contract accounts (CA) for each cluster. Finally, we also determine whether clones inside a cluster tend to be deployed by the same deployer account. To this end, we compute the Gini index for the *number of contract deployments per account*. The Gini index is a measure of inequality [31]. A Gini index of 0% indicates lack of inequality (e.g., when each account deploys the exact same number of contracts). A Gini index of 100% indicates the opposite (e.g., when a single account deploys all the contracts).

Findings. Observation 4) ***93.5% (9,671) of SATD contracts are clones.*** In other words, only 6.5% (668) of SATD contracts do not have a cloned version.

Observation 5) ***The top-10 clone clusters contain together 84.6% of all SATD contracts.*** We identified 934 clone clusters (including singleton ones). Figure 3 depicts the relationship between the *number of clusters* and the *percentage of covered contracts*. The red dashed line in the graph indicates the proportion of SATD contracts that belong to the top-10 clusters. Such a proportion is 84.6%, which indicates that very few clusters concentrate the vast majority of SATD contracts. In particular, the largest cluster contains 69% of all SATD contracts.

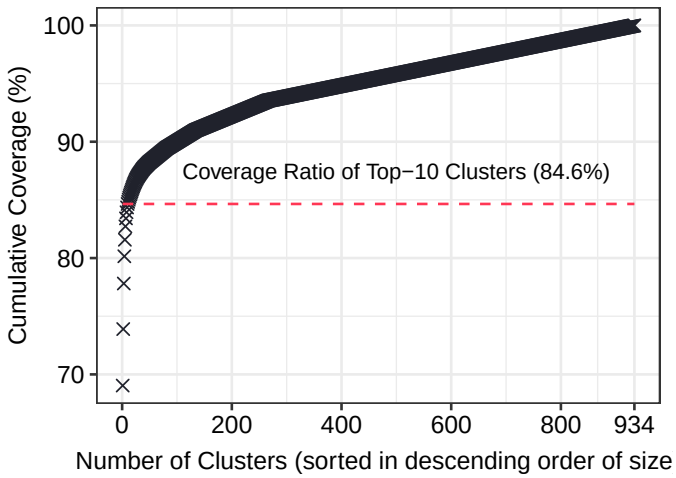

Fig. 3: Cumulative coverage of clone clusters in terms of the ratio of the SATD contracts.

Given the considerable share of the largest clone cluster, we analyze it in more detail. First, we study how homogeneous this cluster is by determining the share of the largest group of identical contracts (a.k.a., type-1 clones) contained therein. More specifically, we compute the `md5` hash of all contracts in the cluster then group those contracts with identical hash. We observe that the largest group of type-1 clones encompasses 99.9% of all contracts in that cluster. That is, almost all contracts within this cluster are identical[12].

11. https://github.com/AlDanial/cloc

12. An example can be found at: https://etherscan.io/address/0x2ebc5330ec52107a654009d0e250f385d969c159#code

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

A manual inspection of this heavily cloned contract (*MultiSig*) reveals that it implements a *multi-signature wallet*. The source code is authored by Gavin Wood, who is an Ethereum co-founder. Further inspection reveals that such a contract was a key component of the *Ethereum Wallet*[13]. The Ethereum Wallet is a now-deprecated *wallet* application also developed by the Ethereum Foundation. When an end-users requests the creation of a multi-signature wallet in the Ethereum Wallet application, the latter creates a copy of the `MultiSig` contract and deploys it using the end-user's EOA. In Appendix C, we describe the other 9 clone clusters in detail.

Observation 6) ***Most SATD contracts have been systematically cloned over time.*** Figure 4 depicts the evolution of the top-10 largest clone clusters. A red dot indicates that at least one contract was created in a given month for a particular clone cluster. Each gold line connects the left-most red dot to the right-most red dot, denoting the duration of cloning for a given clone cluster. Analysis of Figure 4 led us to distinguish between three types of cloning practices:

- Punctual cloning (single month): cluster 8
- Short-term cloning: cluster 4, 5 and 7
- Ongoing cloning: all other clusters. Cluster 1 is clearly systematic (cloning happening almost every month). Clusters 2, 3, 6, 9, and 10 also appear to be ongoing.

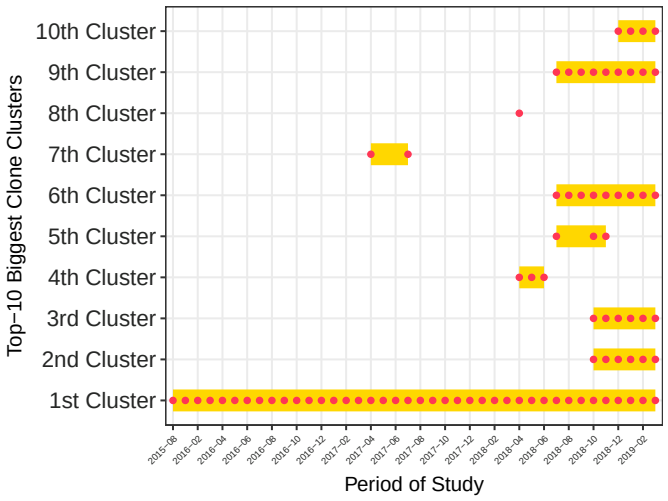

Fig. 4: Evolution of the top-10 largest clone clusters.

Observation 7) ***The majority of clones exist due to cloning performed by other contracts and tools (instead of humans).*** Table 1 displays information regarding deployment authorship per contract cluster. In 9 out of 10 clone clusters, a single type of account is responsible for at least 99% of all deployments. That is, cloning activity in a cluster can almost always be attributed exclusively to either EOAs or CAs. More specifically, EOAs are responsible for the cloning activity happening in four clusters (clusters 1, 5, 7, and 10). In turn, CAs are responsible for cloning activity in five clusters (clusters 2, 3, 4, 6, and 8). In Cluster 9, approximately 70% of the clones are deployed by EOAs. In summary, five clone clusters exist due to CAs. Also, as we discussed before, the contracts in cluster 1 are likely deployed by a tool and not humans (even though the deployer is registered as an EOA). Therefore, we conclude that the vast majority of clones exist due to cloning performed by other contracts and tools (instead of humans).

Finally, we explored whether a single account is responsible for the majority of cloning activities. The results of Gini index are detailed in columns 4 and 7 of Table 1. In summary, the Gini index is greater than zero in all the clones clusters, which indicates that there exists at least one account which has deployed multiple clones of a SATD contract in every cluster. In particular, in clusters 3, 6 and 10, a single account deployed all the SATD contracts.

4.2.1 Code cloning as a confounding factor in empirical studies of SATD in smart contratcs

The observations highlighted so far indicate that SATD is strongly associated with code cloning. In this sense, cloning acts as a major confounding factor in empirical studies of SATD in smart contracts. For instance, if a researcher ignores the cloning aspect while counting SATDs instances in smart contracts, that researcher might be led to believe that there are many more (unique) SATD instances than there actually are. Researchers should thus account for cloning (e.g., remove clones) prior to empirically studying SATD.

In this subsection, we aim to better understand SATD contracts by *explicitly* accounting for code cloning. We perform the following analyses:

- *On the number of SATD instances per contract.* We select a *representative contract* per clone cluster (including singleton clusters). We choose the representative contract to be the one with the highest number of type-1 clones. Finally, we provide descriptive statistics for the *number of SATD instances per representative contract* variable.
- *On the association between the number of SATD instances and contract size.* We investigate the extent to which the size of SATD contracts is associated with the number of SATD instances that appear in those contracts. First, we select representatives using the aforementioned approach. Next, we compute contract size using lines of code (LOC) as given by the Cloc tool (we note that blank lines and comments do not count towards LOC). Then, we calculate the *Spearman's correlation coefficient* between LOC and the number of SATD instances. We choose this correlation coefficient in lieu of others because it does not require the data to be normally distributed.
- *On the domain categories of SATD contracts.* We investigate what domain categories the SATD contracts belong to. First, we take the representative contract of each cluster to form a corpus. Next, we cluster those SATD contracts using a bag-of-words approach and investigate the top-terms per cluster. More specifically, we preprocess the source code (convert to lowercase, remove Solidity reserved keywords[14], remove Solidity's grammar terminals[10], remove numbers, and remove words with less than 3 characters), apply a TF-IDF (Term Frequency - Inverse Document Frequency) [32] vectorizer, apply LSA (Latent Semantic Analysis) [33] for dimensionality reduction, cluster the TD-IDF matrix using K-means [34], and analyze the top-terms (words) per cluster by identifying the clusters' centroids. We employ this simple yet standard pipeline[15] in lieu of topic models

13. https://github.com/ethereum/mist

14. https://docs.soliditylang.org/en/v0.8.19/units-and-global-variables.html#reserved-keywords

15. https://scikit-learn.org/stable/auto_examples/text/plot_document_clustering.html

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

TABLE 1: Contract deployment per type of account in top-10 clusters. Shading indicates the type of account that deployed the highest number of contracts in a cluster.

Top-10 Clusters	Deployer's Account Type					
	External-owned accounts (EOA)			*Contract accounts (CAs)*		
	Contracts Deployed (%)	*Unique Deployers*	*Gini Index for Deployments per Deployer*	*Contracts Deployed (%)*	*Unique Deployers*	*Gini Index for Deployment per Deployer*
Cluster 1	**7,138 (100%)**	**5,323**	**0.24**	0 (0.00%)	0	NA
Cluster 2	0 (0.00%)	0	NA	**502 (100%)**	**404**	**0.18**
Cluster 3	0 (0.00%)	0	NA	**405 (100%)**	**1**	**1.00**
Cluster 4	0 (0.00%)	0	NA	**242 (100%)**	**8**	**0.42**
Cluster 5	**147 (100%)**	**2**	**0.49**	0 (0.00%)	0	NA
Cluster 6	1 (0.84%)	1	1.00	**118 (99.16%)**	**1**	**1.00**
Cluster 7	**71 (100%)**	**5**	**0.72**	0 (0.00%)	0	NA
Cluster 8	0 (0.00%)	0	NA	**54 (100%)**	**2**	**0.44**
Cluster 9	**43 (69.77%)**	**17**	**0.32**	13 (30.23%)	8	0.26
Cluster 10	**31 (100%)**	**1**	**1.00**	0 (0.00%)	0	NA

such as LDA (Latent Dirichlet Allocation) [35] because source code might not fulfill the assumptions of the latter. For instance, LDA assumes that the distribution of topics in a document and the distribution of words in topics are both Dirichlet distributions, which might not hold for Solidity source code.

- *On the prevalence of SATD in smart contracts as compared to that in traditional software.* Prevalence of SATD in traditional software is usually measured as the number of SATD instances found in a given software release. Typically, that number is also compared with the number of total comments found in that release. Translating this analysis to the blockchain domain thus involves investigating DApps. However, properly linking smart contracts to DApps remains an open challenge. For instance, there used to be a website[16] called "State of the DApps", which acted as a hub for blockchain DApps and contained smart contract information. However, this website recently disappeared (404 error). We note that there exists an alternative website[17] called "DApp Radar". Yet, this website puts a paywall for accessing the list of smart contracts of a given DApp. In particular, at the time of writing, a free account is allowed to see the smart contracts from only two DApps, which poses a big barrier for MSR-type studies. Given such a challenge and the focus of this paper, we conduct a best-effort analysis in which we make the rough assumption that a "smart contract always equals an entire application". Next, we take representative contracts, compute the ratio of #SATD/#Comments for each contract, and plot the distribution. Then, we obtain the same ratio for five Java systems by triangulating results from two studies [12, 36]. Finally, we compare the observed ratios for smart contracts and Java systems. Despite the caveat inherently associated with our oversimplification, such an analysis should still shed some light into how the prevalence of SATD in smart contracts compares to that in traditional software.
- *On the SATD indicator patterns and code comment styles.* We analyze the SATD indicator patterns (e.g., `//TODO`) that developers employ to document SATD. Such indicator patterns are provided out-of-the-box by Jitterbug (see Section 3). Next, we analyze the usage of different commenting styles in documenting SATD (e.g., simple single line comments `//`). Our input in this case consists of all textually-unique SATD instances that we observe in our dataset of SATD contracts.

16. www.stateofthedapps.com
17. www.dappradar.com

Findings. **Observation 8)** ***72.4% of representative SATD contracts has a single SATD item.*** In particular, the median and standard deviation of the number of SATD items per representative contract are 1 and 1.45 in turn.

Observation 9) ***There is no association between the number of SATD instances and the size of a contract.*** The Spearman correlation coefficient is 0.02, indicating virtually no correlation. As illustrated by the heatmap depicted in Figure 5, the majority of SATD contracts have a single SATD (regardless of their size).

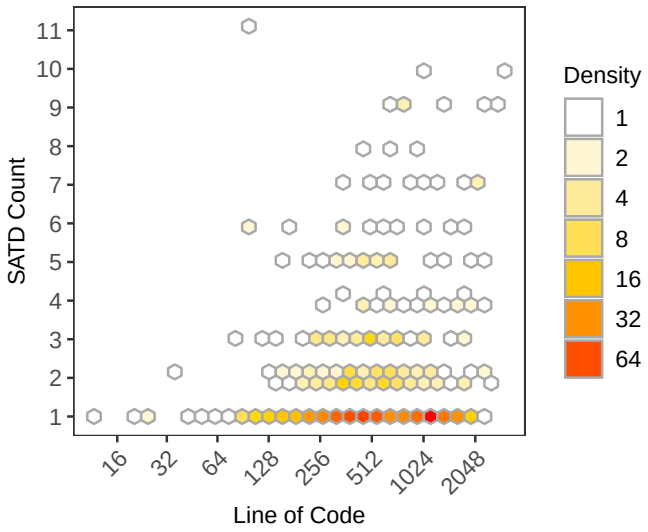

Fig. 5: The correlation heatmap between the number of SATD instances and the size of SATD contracts.

Observation 10) ***SATD appears in contracts that belong to the following five domain categories: blockchain oracles, NFTs token management, role-based access control (RBAC) implementations, token crowdsales, and fungible token management.*** We obtained 5 contract clusters (categories). The top-10 words per cluster are as follows:

- Cluster 1: dynargs args datasource oraclize_query gaslimit oraclizeapi oraclize buf proof self
- Cluster 2: tokenid operator approved index erc721 onerc721received list alltokens ownedtokens exists

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

• Cluster 3: role roles rolename operator whitelist rbac beneficiary balances weiamount file
• Cluster 4: crowdsale state pricingstrategy tokenamount customerid investor receiver weiamount finalizeagent weiraised
• Cluster 5: balances beneficiary success state self index fee block wallet controller

Cluster 1 refers to blockchain oracles. These oracles enable smart contracts to connect to off-chain data (e.g., use of REST APIs) [37]. Cluster 2 refers to contracts that define, manage, and track NFTs (as defined by the ERC721 standard[18]). NFTs are used to identify someone or something in a unique way (e.g., collectible items). In particular, several of the words from cluster 2 appear in the interfaces, contracts, and utilities defined by OpenZeppelin for managing NFTs[19] (e.g., `onerc721received`, originally `onERC721Received`). Cluster 3 refers to role-based access control (RBAC) implementations. RBAC is a popular method to manage access control in smart contracts, as it allows different parties to control who can access different parts of the contract based on business/operations requirements. Cluster 4 refers to token crowdsales, which enable investors to purchase tokens (e.g., NFTs) with Ether. Finally, Cluster 5 refers to contracts that define, manage, and track FT (fungible tokens, as defined in ERC20[20]). Since two examples of the same ERC20 token are exactly the same in type and value, they do not have an id. Hence, the top word in cluster 5 is `balances` (differently from Cluster 2, where the top word is `tokenid`).

Observation 11) ***The #SATD/#Comments ratio is similar between smart contracts and Java applications*** . Table 2 shows the results for the five Java systems that we considered. The ratio ranges from 0.63% to 2.41%.

TABLE 2: Comment and SATD statistics for our five studied Java systems

Project	Release	#Classes	#Comments	#SATD	#SATD/ #Comments
Apache Ant	1.7.0	1,475	21,587	135	0.63%
Apache Jmeter	2.10	1,181	20,084	416	2.07%
ArgoUML	0.34	2,609	67,716	1,630	2.41%
Columba	1.4	1,711	33,895	220	0.65%
JFreeChart	1.0.19	1,065	23,474	247	1.05%

Figure 6 shows the ratio for our dataset of relevant contracts. The interquartile ranges from 0.67% to 1.88%, which is surprisingly close to the aforementioned range from Table 2. Nevertheless, the oversimplification in our approach is potentially inflating the results for smart contracts by not taking into account the other contracts that would be part of a multi-contract DApp. Therefore, our numbers for smart contracts should be interpreted as upper bound for SATD. This analysis should be revisited by future work once smart contracts can be appropriately linked to DApps.

Observation 12) ***Developers mainly employ the TODO pattern in a simple single-line comment (//) to document SATD.*** We observe 555 textually-unique SATD instances. The vast majority of those employ the `todo` pattern. More specifically, `todo` accounts for 494 (89.01%) of all SATD instances, `hack` accounts for 27 (4.87%), `workaround` accounts for 25 (4.5%), and `fixme` accounts for only 9 (1.62%). Finally, we identified that developers employ four different styles of comments to express SATD. Simple single-line (`//`) is the most prevalent style, accounting for 474 (85.41%) of all SATD instances. NatSpec multi-line style (`/** **/`) accounts for 54 (9.73%), simple multi-line accounts (`/* */`) accounts for 18 (3.24%), and NatSpec single-line (`///`) accounts for only 9 (1.62%). The most common combination of pattern and style is `TODO` with a simple single line, accounting for 447 (80.54%) of all SATD instances.

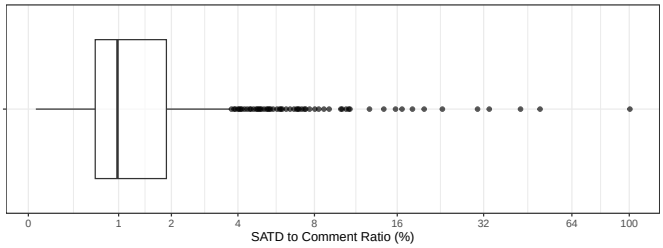

Fig. 6: Distribution of #SATD/#Comments in representative (SATD) contracts.

Summary

RQ2: How does SATD prevalence in smart contracts relate to code cloning?

SATD prevalence is strongly associated with code cloning, as 93.5% of all SATD contracts are clones. In particular:

- The top-10 clone clusters contain together 84.6% of all SATD contracts.
- Most SATD contracts have been systematically cloned over time.
- The majority of clone clusters exist due to cloning performed by other contracts and tools (instead of humans).
- 72.4% of SATD contracts contain a single SATD instance (controlled for clones)
- SATD appears in contracts associated with blockchain oracles, NFTs token management, role-based access control (RBAC) implementations, token crowdsales, and fungible token management.
- The #SATD/#Comments ratio is similar between smart contracts and Java applications
- 80.54% of all SATD instances are expressed with `//TODO` (controlled for clones)

4.3 RQ3: What types of SATD do smart contracts contain?

Motivation. Since there exist different types of technical debt [38, 39], we wish to have a deeper understanding of the types of suboptimal solutions that developers implement and acknowledge. Since blockchain is relatively novel and immature compared to other technologies (e.g., cloud computing), we wonder whether there are SATDs that are inherently associated with Ethereum characteristics (e.g., the gas system). A deeper understanding of SATDs in the blockchain context should serve as a starting point to better assist contract developers in the future, especially in the case of recurrent types of debt. From a contract consumer perspective, debts identification provides increased transparency regarding the quality and expected behavior of contracts, which are crucial in the blockchain domain.

Approach. Similar to Xiao et al. [10], we applied a mixed method that consists of combining open coding and card sorting techniques to build a taxonomy of SATD types. The input to

18. https://ethereum.org/en/developers/docs/standards/tokens/erc-721
19. https://docs.openzeppelin.com/contracts/3.x/api/token/erc721
20. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

this process is a dataset of 555 distinct SATD instances. We obtained such unique instances by removing textually duplicate SATD instances from our dataset. For each SATD instance, we randomly picked a smart contract that contains such SATD to be investigated. Given this input, we planned our qualitative study methodology in three phases (Figure 7): i) Collaborative open coding, ii) Code book evaluation, and iii) Taxonomy creation. In the following, we explain each of these phases in detail:

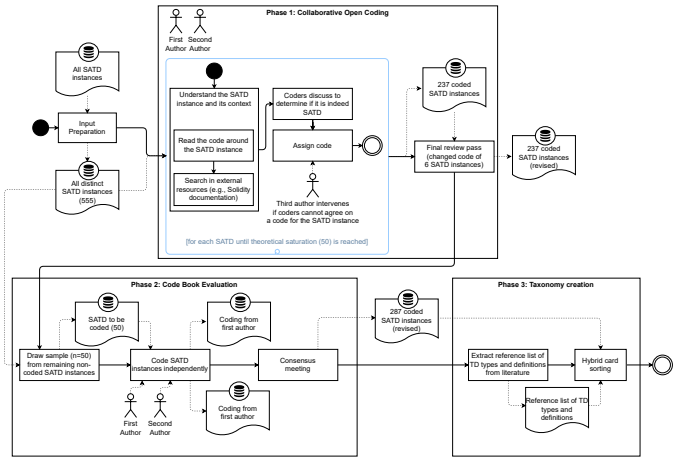

Fig. 7: An overview of our approach for creating a taxonomy of SATD in smart contracts.

i) Collaborative open coding. In this phase, the first two authors (i.e., the coders) collaboratively perform open coding. The first author is a PhD student whose thesis revolves around code quality and upgradability architectures for smart contracts. The second author is a senior researcher who has studied the challenges of developing cost-effective DApps on Ethereum [2, 15, 21, 40–42]. The goal of this phase is to extract a curated set of codes that represent the underlying problems behind SATD instances. To achieve such analytical generalization, we utilize the *theoretical saturation* method [43]. Using this technique, the coders incrementally code randomly picked SATD instances until no new codes emerge. Similar to prior studies [10, 44], we set our saturation threshold to 50. This means that the coders stop the process if and only if no new code is generated upon coding 50 consecutive SATD instances.

Concerning the details of coding, we determined a protocol and strictly followed that for every SATD instance. The coders open the contract that contains a given SATD and rigorously read the context around the line of code where SATD was introduced. Whenever necessary, the coders use external resources (e.g., Solidity documentation, GitHub issue reports, StackExchange posts, etc.) to better the SATD and its context. For example, there were cases in which we needed to understand the Solidity features available in the version with which the contract was compiled. Next, the coders open a discussion to verify that the SATD indeed refers to a sub-optimal solution. Subsequently, if both coders agree on a code, they assign it to that SATD instance. According to the open coding protocol, if they cannot reuse existing codes, they derive a new one and store it in a codebook (available in our online supplementary material package). This process may also result in revising old codes to keep them as mutually exclusive as possible. In situations when both coders are doubtful whether a comment is a SATD at all, they assign *Inconclusive*. If coders recognize that a comment is mistakenly classified as debt by the employed SATD detection tool, they assign *False Positive*. Finally, in case coders cannot reach an agreement for the coding of a given SATD instance, the third author intervenes and solves the disagreement.

We reached saturation after coding 237 SATD instances. After having reached a saturation point, the coders perform a review pass of their code book. This pass ensures coding consistency, because sometimes codes that are created late in the process may better represent early coded SATD instances. Having performed this pass, the coders slightly altered the codes of six SATD instances. Overall, the entire collaborative coding process took 57 man-hours.

ii) Code book evaluation. The goal of this phase is to assess the reliability and generalizability of our coding structure. To this end, we randomly pick an additional sample of 50 SATD instances from the remaining 318 SATD instances that were not coded during the prior phase. Given the list of codes and the coding protocol from phase one, the coders now independently code this new sample. We also consider an extra code called *Other* for situations when a coder recognizes that none of the existing codes are appropriate for a SATD instance. This is followed by a *consensus meeting* where the coders reach a consensus on conflicting cases (12 out of 50). The code chosen by the second coder tended to prevail during the discussion of conflicting cases. This is due to his larger practical and academic experience in the domain of blockchain. All conflicts were solved during the meeting (the third author did not need to intervene). During the meeting, the coders also discuss the *Other* cases and improve the codebook together. To measure reliability, we compute the Cohen's Kappa coefficient to calculate the inter-coder agreement. The resulting coefficient is 0.71, which indicates a substantial level of agreement. This gives us high confidence about the quality of our codes. It is important to point out that we ended up creating only one new code after the consensus meeting, namely *lack of obfuscation*. To investigate the prevalence of this particular code, we searched for the word *obfuscation* in all remaining SATD instances and did not find any match. In total, we coded 287 SATD instances. We note that this number is greater than the size of a statistically representative sample with confidence level and interval set to 95% and 5 respectively (n = 228).

iv) Taxonomy creation. To create our taxonomy, we perform card sorting on our extracted codes. Initially, the first author extracts a reference list of technical debt types and definitions from prior studies conducted in other domains [38, 39, 45, 46]. Then, the first two authors run a hybrid card-sorting meeting during which they organize and abstract codes into major groups and eventually SATD types. We use hybrid card sorting because of two reasons. Firstly, the literature presents solid empirical knowledge on types of SATD in domains other than smart contracts [39, 45, 46]. Secondly, part of this knowledge is directly applicable to smart contracts, since contracts are ultimately just code. Given this, hybrid card sorting enables us to incorporate this knowledge as a basis for grouping codes while giving us authority to generate new groups and SATD types if necessary.

Findings. Observation 13) ***We detected six major types of debt in the smart contract domain.*** Figure 8 shows our taxonomy of SATD types and the number of instances that belong to each category. Of 287 coded instances, 40 (13.9%) and 18 (6.3%) cases were *false positive* and *inconclusive*, respectively. We also had a

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

single conflict that was resolved by the third author accordingly. In the following, we discuss SATD types along with their subcategories. In the following, we discuss each type. Examples are shown in blue.

1. Design debt refers to issues related to the design of a smart contract such as misplaced code, unclear design, non-standard design, or lack of abstraction/encapsulation. We found 16 (5.6%) instances of this SATD class. In the following, we discuss different subcategories.

• *Misplaced responsibility* refers to the issues and tradeoffs made with respect to where developers should locate a piece of code. Such SATDs often imply that a piece of code should be relocated to somewhere else (e.g., libraries) or potentially extracted into a different module. SATDs of this class often violate the single-responsibility principle and contribute to low cohesion, increasing the friction in software maintenance or changeability.

"TODO: Move all insert validation into the DLL lib"

• *Unclear design* refers to developers' uncertainty with respect to various design choices. Doubts regarding the degree of belongingness of a function to a contract, the visibility of variables, or proper identification of contracts to maximize modularity are some examples in this category. In the following example, developers are wondering if they need to create an interface contract rather than the existing solution:

"TODO: - create MonetarySupervisor interface and use it instead?"

• *Non-standard design* refers to issues concerning the lack of adoption of best practices or known standards. For instance, integer overflow and underflow are two disruptive common errors regarding arithmetic operations in solidity. That said, OpenZeppelin released a library called SafeMath that provides safe interfaces for performing such basic operations. In the following example, developers stated to employ the SafeMath contract which is nowadays a well-known accepted solution:

"todo use safemath.sol"

• *Lack of abstraction/encapsulation* refers to the lack of two important software engineering practices. The former employs abstraction to reduce duplication whereas the latter one hides the state of data objects inside a contract or method to avoid unauthorized access. The following two examples highlight the abstraction and encapsulation concerns in turn:

"TODO - you can get a way with a single generic verifier."

"TODO define accessors instead"

2. Documentation debt refers to issues related to improper source code documentation via commenting. We found 4 (1.4%) instances of this SATD class.

• *Insufficient/missing documentation* both refer to the lack of proper documentation at the code level. The first motivating example in Section 2 shows a debt of this class.

"TODO : document"

3. Test debt refers to issues related to the quality assurance (QA) of a smart contract. We found 19 (6.6%) instances of this SATD class.

• *Lack of testing or code reviews* refer to the necessity of conducting (more) unit testing or code review activities with regards to implemented functionalities. For instance:

"TODO: complicated function test it thoroughly!"

"TODO need double check the authority checking"

4. Code debt refers to an umbrella of poor coding practices (a.k.a., smells) such as overly complex codes, naming smells, anti-patterns, workarounds, untrustworthy codes. We found 111 (38.7%) instances of this SATD class.

• *Overly complex code* refers to a piece of code that could have been written in a shorter simplified way. In the following example, developers admitted that the current implementation is complex:

"TODO there must be a shorter way to do this:"

• *Naming smells* refer to issues related to bad naming practices such as non-representative names, spelling errors, etc. In the following example, developers stated that the chosen name does not represent what the function does. Such smells can negatively affect code readability and comprehensibility.

"TODO: consider renaming operator depending on how this is implemented"

• *Anti-patterns* refer to a class of recurrent bad practices in smart contract programming that are often inappropriate, ineffective and risky such as unbounded mass operations and hard coded values.

– *An unbounded mass operation* (a.k.a., unbounded for-loop) [47–49] refers to issues that are caused by having a loop that has no constraint on its number of iterations. Such loops are generally classified as a weakness for smart contracts because they are prone to transaction failure or vulnerabilities. For instance:

"todo Look into making a mass event instead of a one by one if Events use gas"

– *A hard coded value* refers to embedding data (e.g., asset prices, addresses, etc.) directly into a smart contract's source code rather than obtaining it at runtime (e.g., via constructor parameters), making the smart contract a rigid non-reusable software.

"TODO: change addressed before deployed to main net"

• *Workaround code* refers to a temporary, hacky, or shortcut solution that bypasses an existing limitation, bug, or problem. We divided this category based on the origin of the workaround into technology-induced workarounds and other workarounds.

– *Technology-induced workarounds* refer to workaround solutions that are implemented due to limitations of or bugs in the development technologies such as programming languages (e.g., Solidity limitations), frameworks (e.g., Truffle limitations), libraries (e.g., OpenZeppelin limitations), and other third-party limitations.

  ∗ Solidity limitation workarounds *"TODO Remove this library once solidity supports passing a signature to ecrecover. See https://github.com/ethereum/solidity/issues/864"*. See also the second example shown in Section 2.

  ∗ Truffle limitation workarounds: *"FIXME after fix of truffle issue #560: refactor to a separate contract file which uses InvestmentAnalytics interface"*

  ∗ OpenZeppelin limitation workarounds: *"workaround since vault refund does not return refund value."*

  ∗ Other third-party limitation workarounds: *"Hack to get things to work automatically on OpenSea."*

– *Other* refers to other workarounds that are not technology-induced.

"This is a workaround to surpass an issue that Mintabletoken

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

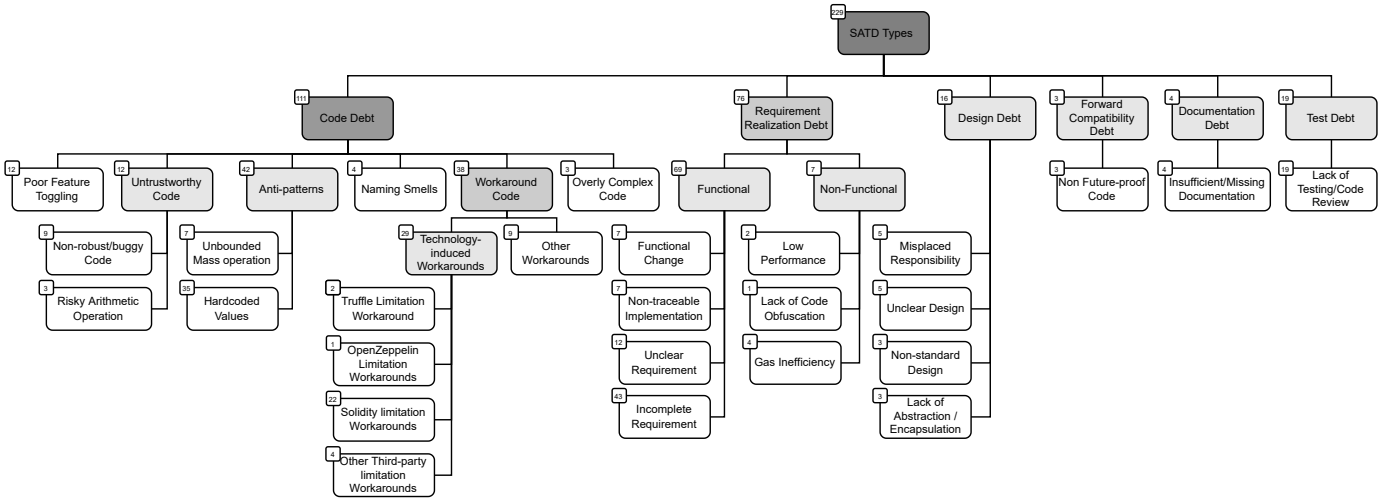

Fig. 8: Our classification of SATD types found in the studied smart contracts. The growing intensity of the rectangles' background color shows how we abstracted our low level codes (i.e., plain rectangles) into major categories and eventually SATD types in each level (i.e., rectangle with similar intensity).

functions are not accessible in this contract. We did not want to change the Openzeppelin code and we did not have the time for an extensive drill down."

• *Untrustworthy code* refers to implementations that are potentially non-robust, buggy, or error-prone.

– *Non-robust/buggy code* refers to implementations that are either non-robust or defective. For instance, codes that cannot handle corner cases.

"TODO: This will introduce an error for very large numbers because the error in our Pi will compound. But for actual reasonable angle values we should be fine."

– *Unsafe arithmetic operation* refers to operations that are prone to overflow or underflow errors and potentially need further tests or refactoring measures.

"TODO - check overflow"

• *Poor feature toggling* refers to a poor coding practice in enabling and disabling features via commenting and uncommenting pieces of codes. For instance:

"TODO: Remove this method before deploying Smart Contract."

5. Requirement realization debt. debt refers to trade-offs related to what requirements should be implemented and how to implement them. We identified 76 (24.5%) instances of this class. We further divided this class into *functional* and *non-functional* subcategories.

• *Functional (requirements)* refers to the services that the contract must provide to its end-users. We observed the following issues:

– *Functional change* refers to situations where the behavior of a piece of code (e.g., function) needs to be changed. For example:

*"TODO: If ether gets accidentally trapped in this contract by some other transaction this function will send it back to the primary in the subsequent order. Change code to only return back what's left over from *this* transaction."*

– *Non-traceable implementation* refers to pieces of code that cannot be easily traced back to requirements, making them subject to removal. For example:

"TODO consider removing. Can this condition be met?"

– *Unclear requirements* refer to developers' uncertainty about requirements or their correctness. For instance, in the following example developers are not sure if they should implement the goalkeeper constraint.

"TODO: Validate that there's at least 1 goalkeeper?"

– *Incomplete requirements* refer to requirements that are missing or partially implemented and need to be completed. For example:

"todo: implement group add someday however func distributeAndReturnToPool gets very complicated"

• *Non-function (requirements)* refer to the quality attributes of a system (as opposed to its behavior). We observed the following issues:

– *Low performance* refers to concerns about the low performance of a smart contract's functionality. For example:

"todo: change this to something faster"

– *Gas inefficiency* refers to concerns with respect to saving gas in smart contracts. This is because a low gas consumption is a critical non-functional requirement for almost any smart contract. Note that an unbounded loop operation can lead to gas-related issues. Nevertheless, we decided to consider it as a separate subcategory under poor code due to its relevance as an anti-pattern [48]. For example:

"TODO optimize for gas usage"

– *Lack of obfuscation* refers to the lack of code obfuscation. Code obfuscation is a method for deliberately creating or modifying an executable source code to prevent others from understanding what it does. For instance:

"Constructor: ToDo: obfuscate"

6. Non-future-proof code refers to situations where the existing implementation can be outdated or suboptimal in the future due to the evolution of surrounding technologies. In the following example, developers implemented a piece of code that checks if a given address refers to a *contract account* or an *externally owned account*. The SATD instance implies that after the Serenity release (a.k.a., Ethereum 2.0, which is planned to be released mid 2022), developers will need to recheck the code because

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

"all addresses will be contracts then". The SATD thus suggests that the existing code is not forward-compatible with Ethereum protocol upgrades:

"XXX Currently there is no better way to check if there is a contract in an address than to check the size of the code at that address. See https://ethereum.stackexchange.com/a/14016/36603 for more details about how this works. TODO Check this again before the Serenity release, because all addresses will be contracts then."

Observation 14) ***Many of our low-level detected SATD categories are unique to smart contracts.*** At the macro level (i.e., the six major types in Figure 8), our detected types comply with a prior study [45] - performed on conventional software - in five cases. We also identified a new type called *forward compatibility debt*, a debt that emerges due to the rapid evolution of blockchain technologies. Similar to [45], we acknowledge that *code debt* (38.7%) and *requirement realization debt* (26.5%) are the most recurrent types of SATD in the smart contract context too. While some micro level categories (i.e., the plain rectangles in Figure 8) are common in other languages (e.g., *lack of testing/code review*, *insufficient/missing documentation*, etc.), we identified many categories that are specific to smart contracts. *Poor feature toggling*, *risky arithmetic operations*, different types of *technology-induced workarounds* and *anti-patterns*, *gas inefficiency*, *lack of code obfuscation*, and *non future-proof code* are inherent in smart contracts.

Observation 15) ***Contract developers sometimes forget to remove a SATD instance once the debt is paid off.*** For a given SATD instance, if we understand that it is already paid off, we categorized it as a false positive. We refer to these code comments as *outdated SATD comments*. Outdated SATD comments represent almost half (43.6%) of all false positives. Outdated SATD comments can mislead both contract authors and consumers (e.g., end-users, other developers reusing the contract), as sometimes it is not trivial to determine whether a SATD comment is outdated. We even observed a few extreme cases in which the piece of code that pays the SATD was located far away from the SATD in the source code (e.g., more than a hundred lines away).

Summary

RQ3: What types of SATD do smart contracts contain?

We identified six types of SATD: code debt (38.7%), requirement realization debt (26.5%), test debt (6.62%), design debt (5.6%), documentation debt (1.4%), and forward compatibility debt (1%). In particular:

- Although our major SATD types are similar to those observed in traditional software [45], several of our low-level SATD types are tied to the domain of smart contracts.
- Practitioners should pay attention to the removal of SATD instances once the debt is paid off to avoid misconception.

5 Discussion

5.1 On the impact of smart contract SATDs on software quality

SATD refers to a suboptimal solution to a problem. That is, developers consciously choose to incur technical debt. Hence, from an internal software quality perspective, we consider SATD to be always bad. However, the actual impact level of a SATD can vary substantially. In particular, it might indeed be the best practical decision sometimes (e.g., when the impact of a SATD is low and it considerably speeds up development time). In certain cases, SATD might even be practically unavoidable.

Understanding the impact of SATD depends on the specification of a target (i.e., impact on what/whom). In this paper, we chose to address the impact of SATD on software quality. This choice is common across technical debt studies[24, 36, 50, 51]. More specifically, by devising the taxonomy shown in RQ3, we can better understand what aspects of software quality are hindered by SATD. For example, we can see from our taxonomy (Figure A) that SATDs impacts the code quality (e.g., due to the use of "poor feature toggling", "anti-patterns", and "workaround code"). Similarly, SATD impacts functional and non-functional requirement realization quality (e.g., code that requires a "functional change" and code with "gas inefficiency") and others. In summary, our results indicate that SATDs impact several dimensions of software quality.

Having said that, we acknowledge that we did not quantify how the decrease in software quality induced by SATDs impacts contract developers, end-users, and other stakeholders. Although clearly fruitful, this research endeavor requires a completely different research setup. For instance, if we focus on contract developers and the "gas inefficiency" SATD type, we would need to estimate how many gas units are being burnt unnecessarily by the function that is being referenced by the SATD (e.g., we would need to reimplement the function in a more gas-efficient way, exercise the function using different input parameters, and compare the number of burnt gas units with the baseline). Alternatively, if we focus on end-users and "gas inefficiency", we would need to estimate how much more end-users had to pay when transacting with the SATD contract in question. Such an estimation depends again on a <baseline, optimized> function pair as well as a computation of the market capitalization of ETH at the time the transaction was sent by the end-users (due to the fluctuation of ETH's market capitalization, which is higher than any fiat currency). In fact, entire papers have been written on the topic of gas inefficiency [22, 52].

We note that certain SATDs seem to impact aspects that are not necessarily associated with the *correctness* of a smart contract. For instance, hardcoded values make the deployment process harder: instead of allowing the values to be changed via constructor parameters, the developers need to modify the source code (likely manually). What if 5 contracts with different values need to be maintained? While all five contracts could be potentially well-tested and even audited, the problem would still hinder (e.g., slow down) the development process. Poor feature toggling is another example of SATD that likely hinders the development process (and yet does not affect correctness).

In the general case, we are afraid that only a proper qualitative study with contract authors (e.g., Solidity experts) and end-users would be able to shed light into the "harmfulness" of SATDs to them (e.g., to their development process). However, such a study is quite challenging from both a methodology and effort perspectives. We recall that contract authors are anonymized in Ethereum (i.e., all we know is the hash ID of the account that created the contract), so it is not easy to discover who the authors actually are. Also, verified contracts in Ethereum commonly do not include the URL to the code repository (e.g., GitHub) where they are being maintained. In fact, it could be the case that the code repository is actually private and not even hosted on GitHub. These constraints make it hard for researchers to track whether/how SATDs are being dealt with,

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

as well as submit PRs that attempt to fix the SATD in order to get feedback from developers. In summary, properly linking verified to authors and GitHub repos is an open challenge. We invite future research to shed light into the practical impact of SATD to contract developers and other stakeholders.

5.2 On the connection between SATD and cloning

From RQ2, we observe a strong connection between SATD and cloning. Such a connection raises several research questions that should be addressed by future research. In the following, we discuss two scenarios:

- **DApps that follow the "one deployment per end-user" paradigm and that contain a contract with SATD.** The *Ethereum Wallet DApp* discussed in RQ2 (cluster 1) is an example of a DApp that follows the aforementioned paradigm (see Observation 5 in Section 4.2). In other words, every time a new user starts using the Ethereum Wallet DApp, a new contract with SATD is deployed. This scenario leaves us with several open questions, including: (q1) does releasing the DApp outweigh the inherent downsides of the SATD?, (q2) is the SATD a minor issue?, (q3) can the SATD contract be easily upgraded, thus rendering the SATD instance less problematic?

- **Frameworks and libraries that contain a contract with SATD.** An example can be seen in the clone cluster 3 mentioned in RQ2. More specifically, clone cluster 3 is composed of identical copies of a single contract called KernelProxy. KernelProxy is a component of AragonOS, which is a smart contract framework for developing robust (e.g., upgradable, permission-controlled) DApps on Ethereum [53]. We thus wonder (q4) whether DApp developers are aware of SATD instances in contracts belonging to frameworks and libraries (dependencies). Assuming that they are aware of the SATD, then (q5) do they fully understand the consequences of the SATD? and (q6) do they use the SATD contract because no better alternative exists (or a better alternative is not known)?

Answering the open questions above (q1-q6) is important, since all the clones that we identified in our paper are relevant, real-world contracts (see Section 3). Yet, answering them is far from trivial. First, it would require designing novel methods for identifying key stakeholders (e.g., the developers behind a DApp that either produces or consumes SATD contracts), since Ethereum accounts do not contain contact information (see Appendix A). Second, it would require designing and conducting an in-depth qualitative study (e.g., an online survey followed by interviews with DApp developers). We thus strongly encourage future research to take on this endeavor, which would shed more light into the interplay between SATD and cloning.

5.3 Is SATD eventually paid off?

Given the several types of SATD that we identified in RQ3 and their associated relevance, we understand that determining whether SATD is eventually paid off would be a natural follow-up work. Such a follow-up is again not trivial, since it would require implementing an automated technique to accurately detect consecutive versions of a given smart contract. We note that Ethereum does have a package manager (EthPM), but it is still in its infancy and most contracts in the wild are not present in such a package manager.

As an initial effort, we conduct a preliminary study in which we estimate the difficulty of paying off SATD by investigating how easy it would be to replace the SATD contract with a patched version. We do so by classifying the upgradability support enabled by each contract (see Appendix A), namely: *no upgradability support* (SATD contract needs to be abandoned, a new version needs to be deployed, and transition is manually done), *rudimentary upgradability support* (SATD contract needs to be selfdestructed, a new version needs to be deployed, and transition often involve manual work), and *upgradability by design* (SATD contract can be automatedly replaced by sending a transaction to a proxy contract).

We proceed as follows. We say that C has *upgradability by design* by analyzing the transactions that C receives. A proxy contract must delegate calls to the logic contract C. Also, we expect C to receive calls from the proxy contract only (except for the transaction that creates C). Hence, we say that C has *upgradability by design* if it received calls from at most two addresses, one of these two addresses is a contract, and such a contract contains the word *(P|p)roxy* in its source code. If *upgradability by design* is not detected, we determine whether a contract has *rudimentary upgradability support*. We say that a SATD contract C has *rudimentary upgradability support* if the source code of C contains the keyword `selfDestruct` or `suicide` (the latter was used in earlier Solidity versions). Finally, if we cannot classify C with either *rudimentary upgradability support* or *upgradability by design*, then we classify C with *no upgradability support*.

Our preliminary investigation indicates that the vast majority of SATD contracts have *rudimentary upgradeability support* (69.8%). Only 1.9% of the SATD contracts have *upgradability by design*. The remaining 28.3% of the SATD contracts have *no upgradability support*. Given the notable size of SATD contracts that implemented the rudimentary technique and the inherent risks of such a technique (see Appendix A), we anticipate that patching SATD can be challenging, and therefore, must be carefully carried out. Future studies should explore if SATD contracts are eventually patched in practice.

We reiterate, however, that these are initial results. Future research should design more robust methods to detect the upgradability mechanisms offered by DApps that use SATD contracts. By studying these mechanisms and other characteristics of DApps (e.g., governance practices), researchers might even be able to build machine learning models that would estimate the likelihood that a given contract (e.g., with SATD) will be replaced in the future. Such a model would not only portray the state-of-the-practice in terms of DApp upgradability, but also inform potential consumers of a DApp about its upgradability.

5.4 Implications to practice

Implication 1) ***Some SATDs are associated with coding anti-patterns. Solutions to those exist.*** anti-patterns are *recurrent* poor programming practices. We identified two categories of such SATDs, namely *hard coded values* and *unbounded mass operations* (Observation 13). The former negatively impacts reusability and is prone to vulnerabilities [54]. Yet, a simple solution exists. It consists of providing the appropriate values through the constructor's or functions' input parameters. In this fashion, developers can configure their contracts by sending a transaction at runtime rather than altering the physical code. The latter anti-pattern is a gas-focused vulnerability, which makes a transaction prone to running out of gas error or exceeding block gas limit error [49] (see Appendix A). We found

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

different solutions to this anti-pattern. The first solution suggests that developers should avoid performing operations for an unbounded number of users. Instead, they should delegate this to users by enabling them to pull from the contract on demand. The second solution is by having an index that tracks the last processed entry before an unbounded loop fails due to the two aforementioned reasons. This enables developers to resume the loop and skip the already processed entries. [49] Finally, breaking a large data structure into smaller chunks and processing it via several transactions can also help not to exceed the block gas limit [55]. More generally, future research should investigate the feasibility of designing and developing IDE plugins (e.g., Truffle plugins) that would detect these recurrent coding anti-patterns, warn users about them, and suggest fixes.

Implication 2) ***Due to the pace with which the blockchain technology evolves, we encourage smart contract developers to design DApps with upgradability in mind.*** Despite the rapid growth of the Ethereum protocol and underlying technologies (e.g., Solidity language, frameworks, and platforms), developers cannot always keep up with demand. More specifically, certain SATDs clearly show gaps between the needs of contract developers and what Ethereum and its ecosystem has to offer. Please refer to *technology-induced workarounds* in Section 4.3 for examples. There is the other side of the coin as well. Due to its rapid evolution, developers are sometimes aware that a given feature in Solidity or in the Ethereum protocol will get superseded, declared obsolete, or become deprecated. Please refer to *non-future-proof code* in Section 4.3 for examples.

In summary, there is a clear friction between developers' needs and the blockchain technology evolution. Both cases discussed above call for designing DApps with upgradability in mind. From a broader perspective, the laws of Software Evolution from Lehman also indicate that every piece of software needs to change to remain useful [56]. As we discuss in Appendix A, there are different approaches to designing upgradable DApps. As a starting point, we suggest that smart contract developers (i) understand the importance of planning for contract upgrades during DApp design time and (ii) choose an upgradability mechanism that best suit their needs and resources. A reference implementation for upgradable contracts is provided by OpenZeppelin [57].

Implication 3) ***Blockchain explorers should flag SATD in contracts.*** We identified six different types of SATD in smart contracts (Observation 13). Contract consumers should be granted with adequate tool support to help them to identify SATD instances in smart contracts. The reason is twofold. First, as we discussed before, contract upgrades are generally difficult to be performed in practice (hence, an inadequate contract choice can have bad maintenance consequences in the long-term). Second, disclosure of the SATDs of a given contract promotes (re)use transparency, enabling contract consumers to reason about how impactful those debts can be for them. In the case where the contract consumer is a DApp developer, knowing and understanding the types of SATD that are present in a third-party contract can influence the design of the DApp itself (e.g., the upgradability mechanism choice). For instance, if a given library contract is not gas efficient, DApp developers might plan to design their DApp in such a way that the library can be easily replaced in the future if needed.

We suggest that Etherscan, the most prominent blockchain explorer, should invest in implementing a feature that (i) detects SATD in the source code of smart contracts and (ii) provides consumers with user-friendly warnings about it (e.g., regarding the type and location of the SATD in the code). As a starting point, Etherscan could search for SATDs by matching the `TODO` keyword in simple single-line comments (Observation 12). In addition, our taxonomy can serve as a reference for SATD type classification (Figure 8). In summary, flagging such SATD contracts enables contract consumers to make an informed decision as to use or not use a given contract.

6 Related Work

6.1 Self-Admitted Technical Debt (SATD)

Self-admitted technical debt (SATD) is a subtype of technical debt. SATD refers to the cases in which developers acknowledge the existence of a technical debt in some software artifact. SATD most commonly occurs in the form of source code comments [11, 58]. In the following, we discuss SATD studies in the context of smart contracts (Section 6.1.1) and traditional blockchain (Section 6.1.2).

6.1.1 Self-Admitted Technical Debt in Smart Contracts

To the best of our knowledge, our paper is the only in-depth study of SATD in smart contracts. In the following, we discuss papers that are still closely related to ours. Ahmadjee et al. [59] design an approach to forecast the cost of deploying a repaired contract that fixes a security technical debt found in the original one. The authors evaluate their approach on 16 smart contracts, which are "either actual contracts identified as vulnerable or explicitly programmed to demonstrate a specific vulnerability." From the results of their experimental evaluation, the authors conclude that their approach allows developers to focus on resolving contract vulnerabilities via technical debt impact analysis and prioritisation. Qu et al. [50] study technical debt in blockchain software (as opposed to smart contracts). More specifically, the authors investigate the following six projects from GitHub: bitcoin (v22), Ethereum (v1.10.12), diem (v1.0.2), solidity (v0.8.10), fabric (v2.3.3), and chia (v1.2.7). Similarly to us, the authors study the prevalence and types of SATD in those projects. The authors observe that SATD is prevalent in all six projects, which contained a total of 1201 SATD instances. The authors classified SATD in seven categories: requirement debt, design debt, documentation debt, compatibility debt, defect debt, test debt, and resource debt (which the authors claim to be a blockchain-software-specific SATD). The most common types of SATD that they observed were design debt and defect debt. Finally, the authors conclude that SATD in blockchain software cannot be entirely captured by focusing only on common SATD tags (e.g., `TODO`). In light of this finding, Qu et al. [60] conduct a follow-up study in which they develop a natural language processing model (BERT [61]) to detect SATD in those blockchain software systems. The authors conclude that their BERT model performs twice as good as a baseline detection method based on SATD tags.

6.1.2 Self-Admitted Technical Debt in Traditional Software

In recent years, the efforts for creating more sophisticated approaches for detecting SATD in software projects has increased among researchers. Potdar and Shihab [11] performed an exploratory study on SATD. Particularly, they analyzed four software projects' comments and extracted a set of 62 handcraft patterns pertaining to SATD. Also, they found that more

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

experienced developers tend to introduce SATD more often than junior ones. Finally, they concluded that code complexity is not associated with the number of SATD (similarly to our study). They also note that only 26.3% to 63.5% of SATD are repaid once introduced in the code. de Freitas Farias et al. [58] proposed a contextualized vocabulary model by utilizing an ontology of software-related terms and mapping these terms to different types of debt. Subsequently, they applied these terms to filter out debt items out of source code comments. In another study [62], the authors observed that, although their initial model [58] could help participants detect SATD instances, the rate of false positive instances was still high. Later on, de O. Passos et al. [63] enlarged the vocabulary model with patterns for the identification of *documentation* and *requirement* debt. Recently, de Freitas Farias et al. [64] proposed an enriched version of the SATD patterns that they had initially extracted [58], and stated that these patterns can be leveraged to support automatic detection and classification of SATD instances.

d. S. Maldonado and Shihab [36] provided a ground truth SATD dataset that was built from a manual inspection of code comments from 5 real-world Java projects. SATD was categorized into the following categories: design debt, requirement debt, defect debt, test debt, and documentation debt. The authors concluded that "the majority of the self-admitted technical debt comments are design debt, which ranged from 42% to 84% across the projects". The authors' definition of *design debt* is broad. They state that design debt comments "[...] indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds or a temporary solution." In our study, we differentiate between design debt and implementation debt. If we were to put those two together, then such a category would account for 53.6% of our manually inspected SATDs. We thus conclude that the most frequent type of SATD in smart contracts is the same one as that found in traditional software. In a subsequent study, Maldonado et al. [24] introduced a Natural Language Processing approach to identify and classify SATD into *design* and *requirement* debt. They trained a maximum-entropy classifier on a corpus of 61,664 comments belonging to 10 open-source software projects. The authors observe that their proposed approach outperformed two baselines: a pattern-based technique and a random classifier. The authors also studied the words that are commonly used to express *design* and *requirement* SATD. Similar to Maldonado et al. [24], Wattanakriengkrai et al. [28] applied machine learning to identify *design* and *requirement* SATD. They used a multi-class Random Forest classifier along with N-gram Inverse Document Frequency (IDF) and under-sampling techniques. The authors compared their proposed classifier with the model developed by Maldonado et al. [24] and observed an improvement of 33% in prediction performance.

Huang et al. [25] implemented an ensemble of machine learning classifiers, each trained on different projects, to determine whether comments of a target project are SATD using a voting strategy. They compared the findings of their work with various baselines and showed that their approach significantly outperforms the baselines. Liu et al. [26] implemented the aforementioned ensemble model as an Eclipse plugin. Recently, Ren et al. [27] proposed a Convolutional Neural Network (CNN) technique to deal with limitations of traditional text mining approaches. Using a CNN along with transfer learning allowed the authors to train a more generalizable model, which can be easily adapted to new projects that would otherwise offer little training data. Concerning the explainability, they found the patterns and features learned by CNN to be more expressive, intuitive, and comprehensive than those summarized by human observations. The CNN also outperforms the latest method proposed by Huang et al. [25] by around 20%. More recently, Yu et al. [12] proposed the Jitterbug framework to identify SATD comments. Jitterbug detects SATD in two steps. First, using a novel pattern recognition algorithm, it detects *easy-to-find* SATD comments with a precision close to 100%. In the second step, Jitterbugs uses a semi-automated machine learning algorithm in a continuous, incremental manner to detect the remaining SATD comments. Their findings show that Jitterbug is more cost-effective in terms of human effort compared to SATD detection approaches that require human intervention.

6.2 Maintenance of smart contracts

While the immutability feature of smart contracts increases their trustworthiness, such a unique characteristic also brings along novel software maintenance challenges. Chen et al. [6] performed a comprehensive survey to explore the main maintenance challenges of smart contracts. The discussion of the challenges was organized according to the traditional software maintenance taxonomy proposed by Swanson [65]. The authors also highlight some common maintenance issues, which include: no perfect contract modification method (e.g., contract replacement, upgradable contracts), high requirement for security and the difficulty in fulfilling it, high requirement for documentation, low code readability (e.g., as hindered by low-level gas optimization code), and lack of experienced developers and researchers.

Our work helps to fill the "lack of advanced SE approaches to remove latent error (e.g., code smell removal and SATD identification)" research gap identified by the authors, as we show that applying a state-of-the-art SATD detector on smart contracts is viable. Our work also helps to fill the "lack of research data" gap, since we design a data collection approach that is able to retrieve the source code of millions of smart contracts.

7 Threats to Validity

Construct Validity. Since Solidity is a relatively new programming language, we could not find any existing well-tested parser that supports our objective of comments extraction. As a result, we had to implement the parser ourselves. While we cannot guarantee that our parser is bug free, we tested it to the best of our abilities.

The vast majority of contracts in Ethereum are of little practical relevance (e.g., toy and test contracts) [2]. To mitigate this bias, we devise a heuristic based on activity level and currency influx to select real-world contracts. Although we carefully craft our heuristic and justify its parameter choices, we cannot guarantee that a better heuristic is nonexistent. We thus invite future research to replicate our study using other heuristics for filtering-in relevant, real-world contracts.

We used a pattern recognition tool called Jitterbug to detect SATD instances. Jitterbug is pre-trained on the code comments of 10 open source software projects [12]. While Yu et al. [12] mentioned that their tool achieved a 100% precision in detecting SATD instances, this performance may not hold in other domains. Our evaluation of false-positive and false-negative

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

rates reported in Appendix B.3 indicates that Jitterbug performs well in the context of smart contracts. We also note that our findings are limited to the four SATD indication patterns that are detected by Jitterbug. Therefore, it is possible that we missed more complex (e.g., less obvious) forms of SATD. We encourage future studies to (i) develop SATD detectors that are specifically tailored for smart contracts (e.g., machine learning models trained on manually labelled smart contract comments) and (ii) investigate more intricate forms of SATD.

We also used the Deckard tool for clone detection in smart contracts. A key parameter in Deckard is the *similarity threshold*. To mitigate construct validity threats, we employed a similarity threshold that has been deemed optimal by a careful sensitivity analysis conducted in a prior study [15].

Internal Validity. To mitigate internal validity threats, we explicitly consider code cloning in Ethereum while studying SATD. In fact, given the strength of the connection between SATD and cloning, we devote an entire RQ to it (RQ2). Except for a few cases that we manually analyzed, we do not establish nor claim *causal* relationships between cloning and SATD. We also mitigate other internal validity threats by ensuring that our chosen statistical analysis procedures properly suit our goals and the characteristics of our datasets.

In RQ3, we classify almost half of false positives as *outdated SATD comments*. Determining whether a SATD is outdated is non-trivial, as it requires technical expertise and possibly domain knowledge. We only classified SATD instances as *outdated* when the two coders mutually agreed on such a classification after a thorough manual inspection of the contract in question. Otherwise, those instances were classified as *inconclusive*. We were thus as conservative as possible.

External Validity. The dataset used in this work is a collection of *all* verified smart contracts that were available on Etherscan as of March 18th, 2019. As such, our dataset is a complete representation of verified contracts at that point in time. Nevertheless, it is possible that smart contracts developed after such a period may have different characteristics. It is thus possible that our results do not generalize to newer contracts.

Different blockchain platforms can have very different characteristics (e.g., regarding centralization, transaction verification protocol, transaction payment mechanism, programming language, virtual machine/runtime, and availability/maturity of IDEs, tools, debuggers, and frameworks). Therefore, it made sense to choose one platform for this study instead of a set of platforms (e.g., to avoid confounding factors in our analyses). Since we had to choose one, we chose Ethereum due to its popularity. Our results are unlikely to generalize to other programmable blockchain platforms.

Since this is the first in-depth study of SATD in smart contracts, we prioritized precision over recall when detecting SATDs. We thus decided to apply Jitterbug's "easy-to-find SATD component", which focuses on the detection of SATDs marked with clear and conventional syntactical structures (e.g., the `FIXME` prefix). As we discussed under Construct Validity, we might consequently be missing less obvious forms of SATDs. Therefore, our results may not generalize to scenarios (e.g., empirical studies) where absolutely all forms of SATDs are detected.

To mitigate external validity threats in our manual classification of SATD types (RQ3), we followed open coding best practices and relied on the notion of theoretical saturation. From a statistics point of view, we highlight that the number of random SATD instances that we analyzed is larger than that of a statistically representative sample (95% confidence level, +-5 confidence interval). In addition, prior to classifying our SATD instances, we carefully reviewed the SATD taxonomies available in the literature [24, 36, 39, 45]. Such a review informed and supported the entire coding process (especially the work of Rios et al. [39]). Finally, we also leveraged our own expertise in smart contracts while performing the classification [2, 15, 21, 40]. We conclude from our manual study that real-world contracts have meaningful and likely impactful debts. Researchers should consider conducting online surveys and interviews to shed more light into the relevance and impact of SATD for different stakeholders (e.g., authors of the contract with SATD, contract developers that reuse a contract with SATD, and end-users).

8 Conclusion

Prior research has legitimately focused on detecting and addressing security vulnerabilities in smart contracts. Nonetheless, being general-purpose computer programs, smart contracts are also susceptible to the maintainability challenges that plague traditional software systems Chen et al. [6]. A key maintainability challenge in software systems is technical debt [7].

In this paper, we conducted a large-scale study of self-admitted technical debt (SATD) in smart contracts. As a first study in the area, we attempted to characterize SATD by determining its prevalence (RQ1), its association with code cloning (RQ2), and its types (RQ3). Despite the immutability property of smart contracts and the difficulty to upgrade DApps, we observed in RQ1 that contracts do sometimes contain SATD (including remarkably relevant, real-world contracts). In RQ2, we noted a strong connection between SATD prevalence and code cloning practices in Ethereum. In RQ3, we derived a detailed taxonomy of SATD types (Figure 8) by manually investigating SATD instances via open coding and card sorting techniques. Based on our findings, we discussed opportunities for future research and implications to practice (Section 5). In particular, we (i) highlight solutions to coding anti-patterns associated with SATD instances that we observed, (ii) encourage developers to consider upgradability early-on during DApp design (based on the misalignment that we noted between developer needs and technology evolution), and (iii) discuss how Etherscan could implement a simple yet effective feature for detecting SATD in verified contracts. We hope that our study will inspire and bootstrap additional maintainability studies on smart contracts, especially in the field of technical debt.

References

[1] D. Furlonger and C. Uzureau, *The Real Business of Blockchain: How Leaders Can Create Value in a New Digital Age*. Harvard Business Review Press, 2019.

[2] 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*, vol. 25, 2020.

[3] "Gartner identifies key emerging technologies spurring innovation through trust, growth and change," Aug 2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-08-23-gartner-identifies-key-emerging-technologies-spurring-innovation-through-trust-growth-and-change

[4] O. Labazova, T. Dehling, and A. Sunyaev, "From hype to reality: A taxonomy of blockchain applications," *Cybersecurity*, 2019.

[5] M. Lacity and R. V. Hoek, "What we've learned so far about blockchain for business," Feb 2021. [Online]. Available: https://sloanreview.mit.edu/article/what-weve-learned-so-far-about-blockchain-for-business/

[6] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintaining smart contracts on ethereum: Issues, techniques, and future challenges," 2020.

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

[7] W. Cunningham, "The wycash portfolio management system," in *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 29–30.

[8] Y. Kashiwa, R. Nishikawa, Y. Kamei, M. Kondo, E. Shihab, R. Sato, and N. Ubayashi, "An empirical study on self-admitted technical debt in modern code review," *Information and Software Technology*, p. 106855, 2022.

[9] M. Vidoni, "Self-admitted technical debt in r packages: An exploratory study," in *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 179–189. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00030

[10] T. Xiao, D. Wang, S. Mcintosh, H. Hata, R. Kula, T. Ishio, and K. Matsumoto, "Characterizing and mitigating self-admitted technical debt in build systems," *IEEE Transactions on Software Engineering*, sep 2021.

[11] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[12] Z. Yu, F. M. Fahid, H. Tu, and T. Menzies, "Identifying self-admitted technical debts with jitterbug: A two-step approach," *IEEE Transactions on Software Engineering*, p. 1–1, 2020.

[13] L. Kiffer, D. Levin, and A. Mislove, "Analyzing ethereum's contract topology," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 494–499.

[14] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," *CoRR*, vol. abs/1905.00272, 2019. [Online]. Available: http://arxiv.org/abs/1905.00272

[15] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and M. Osamu, "Code cloning in smart contracts: A case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, 2020.

[16] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[17] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 06, pp. 42–50, nov 1995.

[18] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, p. 117, Aug 2021.

[19] R. Wang, *Infrastructure as Code, Patterns and Practices: With examples in Python and Terraform*. Manning, 2022.

[20] Ethereum Foundation, "Upgrading Smart Contracts – Upgrade mechanism #3: Proxy patterns," [Online; accessed 10-May-2021]. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/#proxy-patterns

[21] A. A. Zarir, G. A. Oliva, Z. M. J. Jiang, and A. E. Hassan, "Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform," *Transactions on Software Engineering and Methodology (TOSEM)*, 2021, accepted for publication.

[22] B. Severin, M. Hesenius, F. Blum, M. Hettmer, and V. Gruhn, "Smart money wasting: Analyzing gas cost drivers of ethereum smart contracts," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 293–304.

[23] F. Khan, I. David, D. Varro, and S. McIntosh, "Code cloning in smart contracts on the ethereum platform: An extended replication study," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2006–2019, 2023.

[24] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[25] Q. Huang, E. Shihab, X. Xia, L. David, and L. Shanping, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, pp. 418–451, 2017.

[26] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Satd detector: A text-mining-based self-admitted technical debt detection tool," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 9–12.

[27] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, Jul. 2019.

[28] S. Wattanakriengkrai, N. Srisermphoak, S. Sintoplertchaikul, M. Choetkiertikul, C. Ragkhitwetsagul, T. Sunetnanta, H. Hata, and K. Matsumoto, "Automatic classifying self-admitted technical debt using n-gram idf," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 316–322.

[29] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?" in *Annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–3.

[30] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105.

[31] L. Ceriani and P. Verme, "The origins of the gini index: extracts from variabilità e mutabilità (1912) by corrado gini," *The Journal of Economic Inequality*, vol. 10, no. 3, pp. 421–443, Sep 2012.

[32] A. Rajaraman and J. D. Ullman, *Data Mining*. Cambridge University Press, 2011, p. 1–17.

[33] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[34] S. Lloyd, "Least squares quantization in pcm," *IEEE Trans. Inf. Theor.*, vol. 28, no. 2, p. 129–137, sep 2006. [Online]. Available: https://doi.org/10.1109/TIT.1982.1056489

[35] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 993–1022, mar 2003.

[36] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 9–15.

[37] A. Pasdar, Y. C. Lee, and Z. Dong, "Connect api with blockchain: A survey on blockchain oracle implementation," *ACM Comput. Surv.*, vol. 55, no. 10, feb 2023. [Online]. Available: https://doi.org/10.1145/3567582

[38] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology*, vol. 70, pp. 100 – 121, 2016.

[39] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola, "A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners," *Information and Software Technology*, vol. 102, pp. 117 – 145, 2018.

[40] G. A. Oliva and A. E. Hassan, "The gas triangle and its challenges to the development of blockchain-powered applications," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - Ideas, Visions and Reflections (IVR) track.*, ACM, Ed., 2021.

[41] M. Pacheco, G. A. Oliva, G. K. Rajbahadur, and A. E. Hassan, "Is my transaction done yet? an empirical study of transaction processing times in the ethereum blockchain platform," *ACM Trans. Softw. Eng. Methodol.*, jul 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3549542

[42] ——, "What makes ethereum blockchain transactions be processed fast or slow? an empirical study," *Empirical Software Engineering*, vol. 28, no. 2, p. 39, Feb 2023. [Online]. Available: https://doi.org/10.1007/s10664-022-10283-7

[43] K. M. Eisenhardt, "Building theories from case study research," *The Academy of Management Review*, vol. 14, no. 4, pp. 532–550, 1989. [Online]. Available: http://www.jstor.org/stable/258557

[44] T. Hirao, S. McIntosh, A. Ihara, and K. Matsumoto, "The review linkage graph for code review analytics: A recovery approach and empirical study," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 578–589.

[45] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 315–326.

[46] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, "Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, ser. ICSE-SEIS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–10.

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

[48] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, p. 87–95, sep 2020.

[49] N. Kannengiesser, S. Lins, C. Sander, K. Winter, H. Frey, and A. Sunyaev, "Challenges and common solutions in smart contract development," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[50] Y. Qu, T. Bao, X. Chen, L. Li, X. Dou, M. Yuan, and H. Wang, "Do we need to pay technical debt in blockchain software systems?" *Connection Science*, vol. 34, no. 1, pp. 2026–2047, 2022.

[51] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, and Y. Zhou, "How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, jul 2021. [Online]. Available: https://doi.org/10.1145/3447247

[52] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1433–1448, 2021.

[53] "Aragonos intro · aragon developer portal." [Online]. Available: https://hack.aragon.org/docs/aragonos-intro.html

[54] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 327–345, 2022.

[55] J. Chittoda, *Mastering blockchain programming with Solidity : write production-ready smart contracts for Ethereum Blockchain with Solidity*. Packt Publishing, 2019. [Online]. Available: https://books.google.ca/books?id=WB84yAEACAAJ

[56] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *Proceedings of the 4th International Symposium on Software Metrics*, ser. METRICS '97. USA: IEEE Computer Society, 1997, p. 20.

[57] "Using with upgrades." [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/upgradeable

[58] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. d. Silva, and R. O. Spínola, "A contextualized vocabulary model for identifying technical debt on code comments," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, pp. 25–32.

[59] S. Ahmadjee, C. Mera-Gómez, and R. Bahsoon, "Assessing smart contracts security technical debts," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2021, pp. 6–15.

[60] Y. Qu, W. E. Wong, and D. Li, "Empirical research for self-admitted technical debt detection in blockchain software projects," *International Journal of Performability Engineering*, vol. 18, no. 3, p. 149, 2022.

[61] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2023.3289808

bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[62] M. A. de Freitas Farias, J. A. Santos, M. Kalinowski, M. Mendonça, and R. O. Spínola, "Investigating the identification of technical debt through code comment analysis," in *Enterprise Information Systems*, S. Hammoudi, L. A. Maciaszek, M. M. Missikoff, O. Camp, and J. Cordeiro, Eds. Cham: Springer International Publishing, 2017, pp. 284–309.

[63] A. F. de O. Passos, M. A. de Freitas Farias, M. G. de Mendonça Neto, and R. O. Spínola, "A study on identification of documentation and requirement technical debt through code comment analysis," in *Proceedings of the 17th Brazilian Symposium on Software Quality*, ser. SBQS. New York, NY, USA: Association for Computing Machinery, 2018, p. 21–30.

[64] M. A. de Freitas Farias, M. G. de Mendonça Neto, M. Kalinowski, and R. O. Spínola, "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary," *Information and Software Technology*, vol. 121, p. 106270, 2020.

[65] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. Washington, DC, USA: IEEE Computer Society Press, 1976, p. 492–497.

[66] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[67] S. Palladino, "The State of Smart Contract Upgrades," 10 2020. [Online]. Available: https://blog.openzeppelin.com/the-state-of-smart-contract-upgrades/

[68] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, dec 2021.

[69] "Proxies - OpenZeppelin Docs," 2022. [Online]. Available: https://docs.openzeppelin.com/contracts/4.x/api/proxy

[70] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

Amir Mohammad Ebrahimi is currently a Ph.D. student in the Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada. He received his BE in computer Science and Engineering from Ferdowsi University of Mashhad in Iran. He also obtained his Master degree from Amirkabir University of Technology. His research interests include mining software repositories, software quality, and requirement engineering.

Gustavo A. Oliva is an Adjunct Professor at the School of Computing of Queen's University in Canada. Gustavo leads the blockchain research team at the Software Analysis and Intelligence Lab (SAIL). In addition to blockchain, he also conducts empirical studies in the fields of software ecosystems (e.g., npm), software evolution, and explanaible AI. Gustavo received his MSc and PhD degrees from the University of São Paulo (USP) in Brazil under the supervision of professor Dr. Marco Aurélio Gerosa. More information at: https://www.gaoliva.com

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. More information at: http://sail.cs.queensu.ca

Authorized licensed use limited to: Queen's University. Downloaded on July 26,2023 at 16:01:46 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.