APPENDIX A
BACKGROUND

In the following, we describe fundamental blockchain concepts that we use in this paper.

Blockchain. A blockchain is a chronological database distributed and maintained across nodes linked together according to peer-to-peer network architecture. Several blockchain platforms are now being used in practice, including Bitcoin, Ethereum, EOS, POA, Nxt, and Hyperledger Fabric. Ethereum is the most popular *programmable* blockchain platform. A programmable blockchain platform supports hosting and executing *smart contracts*.

Account types. The Ethereum platform supports two types of accounts, namely externally-owned accounts (EOA) and contract accounts. An EOA account contains the following fields: an address (40-digit hexadecimal ID), a transaction counter, and the ETH balance (ETH is the official Ethereum cryptocurrency). A contract account, in turn, holds the bytecode of a smart contract in addition to the previously mentioned fields

Smart contract. A smart contract is a general-purpose computer program. Smart contracts are *immutable*, in the sense that code cannot be patched once deployed. Smart contracts are typically written in Solidity, an object-oriented programming language whose syntax resembles that of JavaScript. Ethereum only stores the bytecode of smart contracts (i.e., the source code is not available in the blockchain itself). A *verified* smart contract is a contract whose source code is available on Etherscan[20]. Etherscan is the most popular block explorer and real-time analytics platform for Ethereum. In this study, we analyze verified contracts only.

Decentralized Applications (DApps) and contract upgrades. Since smart contracts are computer programs, they also have to be maintained [1]. Yet, smart contracts have a distinctive characteristic compared to traditional computer programs: they cannot be patched once deployed. The only solution is to deploy an entirely new contract and redirect traffic to it (contract upgrade). Contract upgrades are typically difficult to be performed [2]. From a technical standpoint, the state of the old contract (e.g., the ETH balance of end-users) typically needs to be migrated to the new contract. From a governance standpoint, the end-users need to be instructed to abandon the old contract and start using the new one. Such a transition can be complicated depending on the size of the user base and how reluctant those users are to changes. Developers can force end-users to use the new contract by destroying the old contract with a call to the `selfdestruct` function. However, not all contracts are implemented with a selfdestruct function. In fact, certain developers strongly discourage the implementation of the `selfdestruct` function due to its possible side effects [3]. Certain DApps are designed from the get-go to support contract upgrades. Sophisticated DApp architectures [4] allow native contract upgrades by (i) redirecting traffic via proxy contracts, (ii) decoupling state from logic, and (iii) allowing end-users to vote whether they want to migrate to the new contract. However, such upgrading capability are relatively new and induce significantly higher testing efforts prior to contract deployment [5].

In a nutshell, DApps can have either (i) *no upgradeability support* (abandon and redeploy), (ii) *rudimentary upgradeability support* (selfdestruct and redeploy), or (iii) *upgradeability by design support* (proxy pattern).

[20]https://etherscan.io/contractsVerified

Transactions and contract deployment. Transactions are the means through which one interacts with Ethereum. EOAs can send transactions to other EOAs to transfer cryptocurrency (Ether). However, since Ethereum is a programmable blockchain, EOAs can also send transactions to (i) deploy contracts and (ii) interact with contracts (i.e., invoke functions defined in deployed contracts). Transactions are always initiated by an EOA. Interestingly, contracts themselves can also deploy and interact with other contracts. These functionalities are enabled by a special mechanism known as internal transactions. Internal transactions are not real transactions, as they are not stored in the blockchain.

When a contract is deployed by an EOA account, it often means that a human (commonly a developer) deployed the contract to Ethereum. When a contract *C2* is deployed by another contract *C1*, it means that *C1* deployed *C2* during runtime as a result of a transaction *t* sent to *C1*. The definition of *C2* can be either given to *C1* (e.g., as part of the data field contained in *t*) or created dynamically by *C1*.

Transaction payment (a.k.a., the gas system). To successfully conduct a transaction on Ethereum, EOA must pay a transaction fee. This fee is a function of (i) the number and type of smart contract instructions that are executed during runtime (a.k.a., the *gas usage*) and (ii) a financial incentive for miners to process the transaction (a.k.a., tip). Higher tips result in faster processing transactions. When sending a transaction, EOAs must also set the *gas limit*, which corresponds to the maximum gas usage that they are willing to pay for. A transaction that exceeds the gas limit results in an *out of gas* error and gets rolled back (but EOA still pays for the failed transaction). More details about the gas system can be seen in [6].

Ethereum block gas limit. In Ethereum, a block corresponds to a set of mined transactions. A block itself are bounded by a specified amount of gas called *block gas limit*. The total amount of gas consumed by all transactions in the block must be less than the block gas limit. Consuming more gas results in *exceeds block gas limit* error.

APPENDIX B
SATD DETECTION

B.1 Choice of SATD detector tool

Table 3 shows an overview of the candidate SATD detector tools that we considered based on a literature review. In order to choose an appropriate SATD detector tool, we study and compare multiple criteria among several studies that proposed a detection approach. To begin with, the availability of the replication package is a primary factor in selecting our target SATD detector. This is because re-implementing existing approaches can be complicated, error-prone, and time-consuming. Out of seven studies, four studies released their tool replication packages [7–10]. We excluded the remaining three studies [11–13]. The next factor is concerned with the purpose of the tools. Since we are interested in detecting SATD in general, we exclude tools that identify specific types of SATD. All four tools determine SATD versus non-SATD comments. Hence, none of them are excluded after applying this criterion. Our next filter is concerned with the tools' approach. In terms of approach,

these four tools can be classified into three categories: i) pattern-based techniques, ii) machine learning (ML) or natural language processing (NLP) techniques, iii) a hybrid of the first and second techniques. In the following, we discuss each category along with our rationale for selecting or excluding each approach.

Potdar and Shihab [7] handcrafted a curated list of 62 SATD indicator patterns. While such patterns can be applied for SATD detection, a fully automated detection approach based on these patterns can result in a high false-positive rate. For instance, Bavota and Russo [14] used a similar list of SATD patterns and mentioned that over 25% of their manually investigated SATD comments were false-positive. Given this finding and the highly descriptive nature of smart contract comments, we prefer not to employ this technique.

Huang et al. [8] proposed a voting mechanism that is based on training multiple machines learning sub-classifiers. Later on, Later on, Liu et al. [9] implemented a plugin to integrate this approach in the Eclipse development environment. We decided not to use this tool because Yu et al. [10] mentioned that they found several comment instances that were misclassified in n Liu et al.'s study [9]. This can negatively affect the classification model trained by this work.

Yu et al. [10] introduced a hybrid two-step framework called Jitterbug. In the first phase, they applied a novel pattern recognition algorithm for detecting easy-to-find SATD comments. Subsequently, they implemented a semi-automated continuous learning technique in the second phase to identify forms of SATD that do not contain any SATD indicator patterns, namely hard-to-find. We decided only to use the easy-to-find detection component. The reasons are threefold. Firstly, in our work, we strive for precision rather than recall, and it was shown that this component was practical with 100% precision in detecting SATD comments in ten open source projects. That said, we also evaluate and discuss the performance of this component in the smart contracts context in Appendix B.3. Secondly, we do not know the precision of the hard-to-find SATD detector (i.e., the second step of the framework), and evaluating this component is out of the scope of our paper. Instead, our goal is to reuse a reliable SATD detector. Finally, given the semi-automatic nature of the hard-to-find component, it was not feasible to apply it to our dataset with over 17M comments.

B.2 Training dataset for Yu's SATD Detector

Table 4 describes the dataset in terms of the selected software systems, the domain of each system, and the number of comments as well as SATD comments. This corpus contains 62,275 comments and 3,871 SATD comments of the 10 open-source software projects.

B.3 Evaluation of the employed SATD detector tool

To evaluate the performance of our employed SATD detector tool, we evaluate its false-positive (FP) and false-negative (FN) rates.

To calculate the false-positive rate, we leverage the manual analysis conducted as part of RQ3 (Section 4.3). As part of that analysis, we investigated a statistically representative sample of 287 SATD comments and counted the number of code comments that were incorrectly classified as SATD by Jitterbug. We identified 40 (FP = 13.9%) false-positive cases. However, 42.5% of those were classified by us as false positives because the debt mentioned in the comment had already been paid off (i.e., outdated SATD comments). It is worth mentioning that none of the existing SATD detection tools can distinguish between paid and non-paid SATD.

Furthermore, we assess the false-negative (FN) rate of Jitterbug's easy-to-find detector tool. First, we drew a statistically representative sample of code comments (95% confidence level with a confidence interval of +-5) from the set of textually-unique comments that are classified as non-SATD by Jitterbug. We obtained 383 code comments. Next, the first two authors manually studied the sampled comments and classified them as either SATD or non-SATD. We only identified 2 (FN = 0.5%) code comments that were actual SATD instances. Such a result suggests that Jitterbug has a very low FN rate.

APPENDIX C
LARGEST GROUP OF TYPE-1 CONTRACTS PER CLONE CLUSTER

Given the number of contracts inside the top-10 largest clusters, we decided to investigate these clusters in more detail. We refer to the largest clone cluster as cluster 1, the second-largest clone cluster as cluster 2 and so forth. We first analyze how homogeneous these clusters are by determining the share of the largest group of identical contracts (a.k.a., type-1 clones) within each cluster. We discover the groups of identical contracts within a cluster by calculating the md5 hash of contracts and then grouping contracts with identical hash. Our results indicate such a share is 100% for clusters 2,3 and 8, 99.5% for cluster 1, 99.2% for cluster 6, 98.6% for clusters 5 and 7, 74.4% 71.0% and 29.3 for clusters 9, 10 and 4 in turn. (Table 5). In other words, for the majority of clusters, the contracts contained therein are mutually identical (type-1 clones).

TABLE 5: Distribution of type-1 clones over the top-10 clones clusters.

Top-10 Clusters	Number of groups of type-1 clones	Total number of contracts	Share of the largest group of type-1 clones
Cluster 1	10	7138	99.5%
Cluster 2	1	502	100.0%
Cluster 3	1	405	100.0%
Cluster 4	6	242	29.3%
Cluster 5	3	147	98.6%
Cluster 6	2	119	99.2%
Cluster 7	2	71	98.6%
Cluster 8	1	54	100.0%
Cluster 9	6	43	74.4%
Cluster 10	2	31	71.0%

Given how homogeneous the top-10 clusters are, we pick a *representative* contract from each cluster and briefly describe its purpose in order to describe the cluster. Such a description will give insights into the types of SATD contracts that are frequently cloned. The representative contract of a cluster C is any contract that belongs to the largest group of identical contracts within C.

• Cluster (1): The contract was written by Gavin Wood (Ethereum co-founder) and it implements a *multi-signature wallet*. In simple terms, a multi-signature wallet contract implements a cryptocurrency wallet that

TABLE 3: Overview of the candidate SATD detector tools.

Study	Year	Venue	ML/NLP?	Pattern-based?	Automation Level	Target Debt	Tool Avail.?
Potdar and Shihab [7]	2014	ICSME		✓	Automatic	SATD, Non-SATD	✓
Maldonado et al. [11]	2017	TSE	✓		Automatic	Design, Requirement, Non-SATD	
Huang et al. [8]	2018	EMSE	✓		Automatic	SATD, Non-SATD	✓
Liu et al. [9]	2018	ICSE	✓	✓	Automatic	SATD, Non-SATD	✓
Ren et al. [12]	2019	TOSEM	✓		Automatic	SATD, Non-SATD	
Wattanakriengkrai et al. [13]	2019	APSEC	✓		Automatic	Design, Requirement, Non-SATD	
Yu et al. [10]	**2020**	**TSE**	✓	✓	**Semi-automatic**	**SATD, Non-SATD**	✓

TABLE 4: Summary information of the dataset that we used to train Jitterbug (c.f., Section 3)

Project	Domain	Comments	SATD (Ratio)
JMeter	Testing	8,057	374 (4.64%)
Columba	Email Client	6,468	204 (3.15%)
Hibernate	Object Mapping Tool	2,968	472 (15.90%)
JFreeChart	Java Framework	4,408	209 (4.74%)
SQuirrel	Database	7,215	286 (3.96%)
Apache Ant	Automating Build	4,098	131 (3.2%)
ArgoUML	UML Diagram	9,452	1,413 (14.95%)
EMF	Model Framework	4,390	104 (2.37%)
JEdit	Java Text Editor	10,322	256 (2.48%)
JRuby	Rubby for Java	4,897	622 (12.70%)

requires the confirmation of multiple people (i.e., multiple signatories) before the funds can be transferred from/to the wallet. Contract address: https://etherscan.io/address/0x6ae112b39805d5d4a1c807c2d8e8c554607ee110#code

• Cluster (2): The contract implements the *Proxy* design pattern [15], enabling internal dependencies to other contracts to be reconfigured during run-time (e.g., enabling *upgradeable contracts*). Contract address: https://etherscan.io/address/0x538fae85a5aad094148bee8b75643d2f0df4c864#code

• Cluster (3): This contract is a part of AragonOS[21] framework that provides developers with a various contracts for building decentralized organizations, DApps, and protocols. Contract address: https://etherscan.io/address/0x3b20e426413ec06cedf1674e1bb671b6bf8cfd9b#code

• Cluster (4): This contract implements core functionalities for a decentralized gambling application. Contract address: https://etherscan.io/address/0xd9e8a52f712b0273770581f33b766a76724408d3#code

• Cluster (5): This contract is a component of an end-to-end decentralized betting application. In particular, this application enables end-users to place crypto bets on different sports and esports events. Contract address: https://etherscan.io/address/0x8021338f306fea03cf31528561efdb83365c91bb#code

• Cluster (6): This contract implements basic functionalities for a decentralized exchange application where end-users can buy or sell crypto-currencies or tokens. Contract address: https://etherscan.io/address/0xdf72b12a5f7f5a02e9949c475a8d90694d10f198#code

[21] https://hack.aragon.org/docs/aragonos-intro.html

• Cluster (7): This contract is similar to the multi-signature wallet in Cluster 1 with severallight modifications. Contract address: https://etherscan.io/address/0x05276c19f670ffe7ba6b3bdc94e70c36b4b9dad5#code

• Cluster (8): This contract is also a follow-up version of the discussed contract in Cluster 4: https://etherscan.io/address/0x1d896c9da6a32245cd035699dedaffd7f06f832c#code

• Cluster (9): This contract combines an upgradeability proxy with an authorization mechanism for administrative tasks: https://etherscan.io/address/0x6de037ef9ad2725eb40118bb1702ebb27e4aeb24#code

• Cluster (10): This contract is a standard ERC20 token application. Such Tokens are often considered as assets and end-users can invest in. Contract Address: https://etherscan.io/address/0x4204c9d1546c7a1394faad62f1ff249d355020e3#code

Appendix References

[1] 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.

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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

[9] 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.

[10] 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

[11] 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.

[12] 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.

[13] 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.

[14] 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

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