Maintenance of Long-Living Smart Contracts

Matthias Lohr
CCRDMT, University of Koblenz-Landau
Koblenz, Germany
matthiaslohr@uni-koblenz.de

Sven Peldszus
University of Koblenz-Landau
Koblenz, Germany
speldszus@uni-koblenz.de

***Abstract*—In recent years, blockchains became widely known for offering immutable and trust-free storage of arbitrary information. Blockchains also leverage smart contracts, a concept for executing program code for modifying the blockchain state. While the characteristics of a blockchain, especially immutability, enable reliability in a trust-free environment, it hinders the maintenance of smart contracts itself.**

With the increasing number, size, and lifetime of smart contracts, they could be considered to be long-living software. Therefore, it might become necessary to apply common techniques from software engineering to maintain smart contracts. However, updating smart contracts residing within the immutable blockchain data raises an interesting challenge.

In this work, we analyze whether the assumption of smart contracts being long-living is true, study how immutable smart contracts are maintained in practice and elaborate the challenges appearing due to these maintenance practices.

***Index Terms*—blockchain, smart contract, software engineering, maintenance, security**

I. Introduction

Blockchains are considered to offer an immutable storage for arbitrary data in a decentralized manner. In addition to the widely known use case as crypto currency, it is also possible to store the code and the result of executable programs called smart contracts. Due to the property of immutability of blockchains and thus also of the smart contracts stored, and due to the determinism of the execution of smart contracts, it is possible to validate execution results, which is the underlying concept of a blockchain. The immutable storage of smart contracts on a blockchain makes the contracts available forever. The fact of smart contracts being available and executable forever might qualify them as long-living software. However, a system is not necessarily long living only because of its implementation being stored and executable over a long period of time. There are additional factors like maintenance and active usage that have to be considered.

One of the biggest issues most long-living systems in the software context have to deal with, is structural decay due to software aging [1] resulting in the formation of design flaws [2]. Usually, long-living systems are subject to continuous changes due to bug fixes, adoption to changing environments, or extension with new features. When using the blockchain for storing smart contracts, resulting from the immutability property, it is not possible to update or delete smart contracts. Therefore, there is no chance of fixing bugs or adding new features once a smart contract is deployed to the blockchain. However, this implies that a smart contract has to be bug free and feature complete starting from the first time it is deployed on the blockchain.

While the two challenges of implementing bug free and feature complete smart contracts could theoretically be solved for very small and finalized functionalities, changes in the environment might require a change in the software system but can neither be foreseen nor planned. These environmental changes might include legal changes (e. g. GDPR [3]), changes of underlying system (e. g. changes of how smart contracts are executed on the respective blockchain instance), new attacks against security mechanisms that have been assumed to be secure (e. g. DES [4]), etc. To keep the smart contract legal or secure it has to be changed.

In this work, we discuss the challenges in maintaining long-living smart contracts. We first elaborate the assumption that smart contracts can be considered as long-living software. We then discuss, which mechanisms are available to update smart contracts and how developers can deal with the urge to update smart contracts.

The remainder of this work is structured as follows. At first, we provide a brief background on blockchain and smart contracts in Sec. II. In Sec. III, we discuss whether smart contracts can be considered as long-living software, the update mechanisms provided by the different smart contract platforms, and how smart contracts are updated in practice. Afterwards, we discuss related work in Sec. IV and conclude in Sec. V.

II. Background

For a better understanding of the concerns of this paper, we provide some general and some blockchain environment specific background information on smart contracts.

A. General Background on Smart Contracts

In the context of blockchains, a smart contract is a software program which is executed by several nodes participating in the respective blockchain network. A smart contract contains instructions on how the state of the blockchain should be modified depending on the set of provided input parameters. Furthermore, depending on the actual implementation, it is also possible for a smart contract to trigger external actions which are not observable by the blockchain. Since on-chain state changes are observable by the blockchain, i. e. by all participating nodes, each node can validate and accept or ignore proposed state changes if their own computation result equals or differs to the previously published result.

One of the main properties utilized by smart contracts is the concept of being free of trust. As blockchains allow us to trace everything that happened on a blockchain, there is no need to trust someone as we can proof everything ourselves.

There are different approaches for the implementation of smart contracts. In this paper, we focus on the most important blockchain platform in terms of market capitalization[1], the Ethereum blockchain [5]. To give an impression about the possible variety of implementations, we compare Ethereum to Hyperledger Fabric [6], a blockchain concept which significantly differs from Ethereum in several properties, but supports Turing-complete smart contracts as Ethereum does.

In the following subsections we give more background information about Ethereum and Hyperledger Fabric and describe identified properties regarding smart contract implementation on blockchains.

B. Ethereum Smart Contracts

Bitcoin[7], the crypto currency that made blockchains became widely known, already offered support for a simple set of rules for unlocking account balances. Ethereum [5] is the second most important public blockchain platform in terms of market capitalization after Bitcoin, but the most important one that offers support for Turing-complete smart contracts. In Ethereum, smart contracts can be written in *Solidity*[2] which is compiled into bytecode. For deployment, the bytecode is written to the Ethereum blockchain (using a *create* transaction). A deployed smart contract can be identified by its address, which is calculated using a cryptographic hash function. Once a smart contract is deployed, Ethereum does not allow for modification or deletion of the smart contract code. However, according to the rules defined in the smart contract, it is possible to modify the state of the smart contract by invoking smart contract methods (using *call* transactions). Ethereum smart contracts can implement a *selfdestruct* method, which allows the owner to mark a contract as obsolete. Selfdestruction results in not being able to use the smart contract anymore and is not reversible.

When a user wants to call a smart contract method, he creates a *call* transaction with the address of the Smart Contract and all input parameters. This transaction is published to the network and processed by so-called mining nodes, which will execute transaction and create a new block with the outcome. Since the order of the transaction and the bytecode of the smart contract is publicly known, everybody *can* validate the transaction. For including the result into a new block, miners *have to* execute the code. Besides the inherent consensus algorithm of the Ethereum blockchain there is no need for conflict resolution for transaction results, since – assuming correct execution – each mining node will come to the same result. This way, state changes are deterministic and unambiguous. The execution pattern of Ethereum is called *Order-Execute*.

For our analysis, we used Ethereum data dumps provided by Blockchair.com [8]. Since the Ethereum main network was started in July 2015 until end of October 2019 2,988,133 contracts have been deployed so far. According to Fig. 1 the number of created and executed smart contracts per day has continuously increased. The same holds for the size of the smart contracts, which is visualized in Fig. 2.

[1]CoinMarketCap Ranking – http://coinmarketcap.com [accessed Dec19]
[2]Solidity Documentation – http://solidity.readthedocs.io [accessed Dec19]

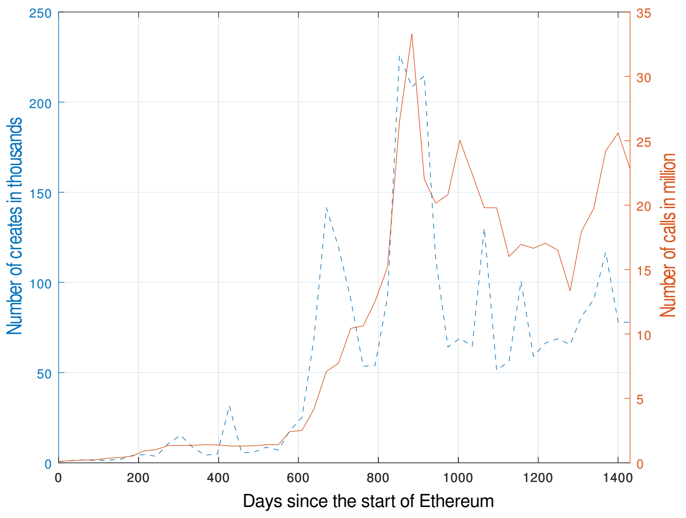

Fig. 1. Number of calls (solid line) and creates (dashed line) over time

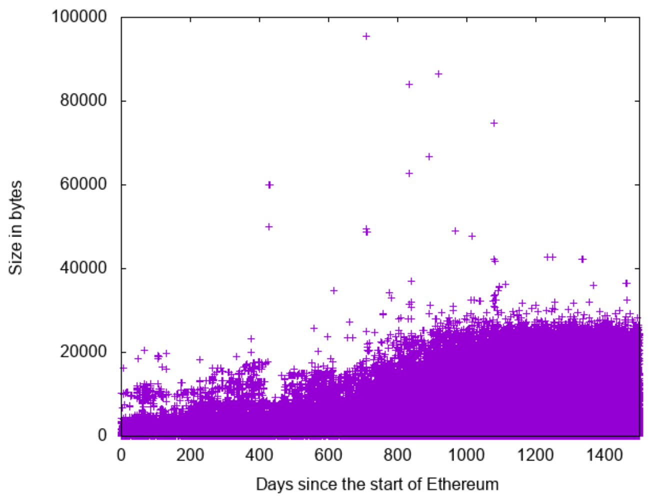

Fig. 2. Contract size over time

C. Hyperledger Fabric Smart Contracts

Hyperledger Fabric [6] is a framework for creating business grade permissioned blockchains. The main differences between permissioned and public blockchains are that a permissioned blockchain requires authorized access to the blockchain while public blockchains allow users to create an unlimited number of pseudonyms. In Hyperledger Fabric, smart contracts can be written in any programming language and run as standalone software. The code of a smart contract is not part of the blockchain but can be developed and distributed using established software deployment and delivery architectures. When a smart contract program is started, it

TABLE I
COMPARISON OF SMART CONTRACT PLATFORM CHARACTERISTICS REGARDING ETHEREUM AND HYPERLEDGER FABRIC

	Property Name	Ethereum	Hyperledger Fabric
a)	Programming Language	specific	arbitrary
b)	Code Deployment	on-chain	custom
c)	Code Distribution	on-chain	custom
d)	Executing Peers	mining and validating nodes	endorsing nodes
e)	Execution Environment	Ethereum Virtual Machine	arbitrary
f)	External Communication	not possible	possible
g)	Result Distribution	on-chain	on-chain

listens to incoming transactions, calculates the result according to the input parameters and the code and sends back the result to the blockchain infrastructure.

When a user wants to call a smart contract, he creates a new transaction containing the name of the smart contract and all input parameters. This transaction is published to the network and processed only by nodes having access to the smart contract program (called *endorsing nodes*). All endorsing nodes send their results to an *ordering service*, which defines the order in which the transactions are applied to the blockchain. After the transaction results have been ordered, each node in the network applies the outcomes following predefined policies for that specific environment. These policies can define, how many and which kind of nodes have to present a certain result before it gets accepted to be appended to the blockchain.

D. Smart Contract Platform Characteristics

We identified the following characteristics regarding smart contract execution. For a comparison between Ethereum and Hyperledger Fabric see Table I.

a) Programming Language: Depending on the smart contract platform, smart contract developers are limited in choosing programming languages.

For example, Solidity was influenced by existing programming languages like C++, Python and JavaScript, but is designed for use with Ethereum only. In Hyperledger Fabric, a smart contract runs as individual application, so it is possible to use any programming language for software development.

b) Code Deployment: Code deployment denotes how the blockchain network is informed about a new smart contract. Other nodes can use this information for creating transactions to request smart contract execution.

In Ethereum, this is done by creating a transaction which contains the smart contract and registers it with the blockchain. In Hyperledger Fabric, this is done by publishing some meta information (smart contract name, smart contract version, ...) to the network. The actual code is not part of this information.

c) Code Distribution: This property describes, how the smart contract code is distributed to the nodes which want to execute the smart contract.

In Ethereum, since the smart contract code is deployed to the blockchain, it is distributed along with the block data. In Hyperledger Fabric, distribution is up to the smart contract operator, which can use common software distribution mechanisms (e. g. a website, package repository, etc.).

d) Executing Peers: In a blockchain network, executing smart contracts could be a bottleneck if all nodes are obliged to execute any smart contract, e. g. for verification. The whole network would have to wait for the slowest node. For this reason, only a subset of nodes in a network is actually executing smart contract code.

In Ethereum, since the smart contract code is part of the publicly available blockchain data, any node could execute smart contracts. But only the one node, which “finds” a new block is getting paid for executing smart contracts by receiving transaction fees. Other nodes, which want to verify the execution result can, but do not have to, execute the smart contract on their own. In Hyperledger Fabric, nodes can, depending on the smart contract, have the *endorser* role. Only endorsing nodes will execute smart contracts. However, before a node can become an endorsing node, the node operator has to get the smart contract code.

e) Execution Environment: The execution environment defines the limits in which a smart contract can run regarding both functionality and resource consumption.

In Ethereum, smart contracts are executed in the *Ethereum Virtual Machine* (EVM), which supports Turing-complete programs provided, but with a limited subset of available libraries. Each bytecode instruction has an assigned cost value (*Gas*), which has to be payed for executing the smart contract. This way, the Ethereum Virtual Machine protects executing nodes from high computational load, since the execution will abort if the Gas limit is reached. In Hyperledger Fabric the execution environment is not constrained. Nevertheless, smart contract code is usually executed in a virtual environment (e. g. Docker) for protecting the host from possible malicious actions. However, it is still possible to induce high loads to the host system, if the virtual environment is not properly configured for limiting available hardware resources.

f) External Communication: Depending on the application, a smart contract may need additional data, which is not provided by the contract call. In this case, a smart contact needs access to external data sources (e. g. weather sensor).

In Ethereum, it is not possible to query off-chain data sources. Instead, it is possible to create so-called *Oracles*. An Oracle is a smart contract, which contains a certain information and can be updated from another source using smart contract method calls. However, the process for updating the information contained in the oracle can not be initialized by a smart contract. In Hyperledger Fabric, since a smart contract is realized as a standalone software program, it can access any data source it wants to. Depending on the data source (e.g. random value generator) this might result in different results for multiple smart contract invocations with the same input parameters provided in the call transaction. Therefore, a method for result conflict resolution is required.

g) Result Distribution: Smart contract execution would be useless if the execution results are ignored. Therefore, all blockchain environments known to us use the blockchain as a log for the results.

On Ethereum, every node can validate the result since input parameters, code and result are written to the blockchain. In Hyperledger Fabric, predefined policies decide how consensus is reached if endorsing peers offer different possible results.

III. Discussion

In this section, we discuss the state of the art regarding the usage and update mechanisms of smart contracts. This covers an analysis whether smart contracts can be considered as long-living software and a summary of the update mechanisms provided by different smart contract platforms. From the state of the art and practical usage of update mechanisms we derive issues in the development of long-living smart contracts and discuss how software engineering can address those issues.

A. Are Smart Contracts Long-Living Systems?

As we described before, the immutability of the Ethereum blockchain seems to automatically result in every smart contract to be long-living as it cannot be removed or updated. However, only being stored on the blockchain does not necessarily imply longevity. In the context of smart contracts we consider longevity as the active use of a smart contract over a long period of time. However, even for traditional long-living software systems this period of time is not precisely defined [9]. As smart contracts are a comparably young technology we cannot consider the long period of usage in terms of decades like e. g. the software used by financial institutes. For this reason, the goal is to study whether the current usage of smart contracts indicates a potential use similar to traditional long-living software.

To study the usage and longevity of smart contracts we analyzed the Ethereum blockchain between its creation in mid 2015 and end of October 2019 (in total 1,555 days) using the data provided by Blockchair [8]. All in all, this time frame covers the creation of 2,988,133 smart contracts of which 1,048,835 smart contracts have been called at least once. 606,821 smart contracts have been called more than once. For the following discussion, we excluded the smart contracts that have never been called.

To answer the question whether smart contracts are long-living, we calculated the lifespan of each contract created by determining the time between the creation of the contract on the blockchain and its last execution.

In Fig. 3 we plotted how many smart contracts have been deployed that reach a certain lifespan (measured in days). We can see that many smart contracts are used immediately after

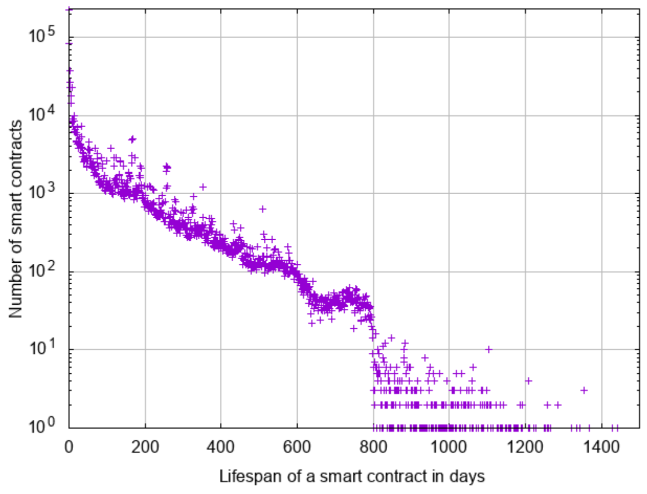

Fig. 3. Lifespan of smart contracts

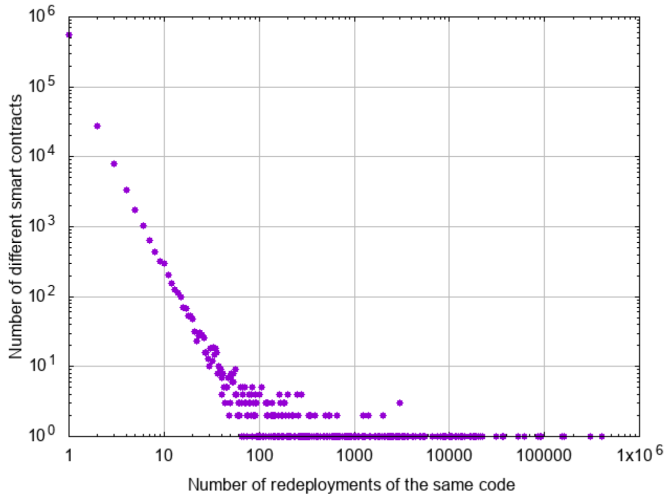

Fig. 4. Amount of contracts that have been redeployed x times

creation and get inactive after a few days. The majority of smart contracts has a lifespan between a couple and about 800 days. Nevertheless, there are still many smart contracts with a lifespan higher than 800 days up to being alive for nearly the entire time frame considered. Those smart contracts might be considered as long-living.

Also smart contracts which have been used for a short period of time could be considered as long-living if they are deployed multiple times, distributed over a longer period of time. For this reason, we counted how often exactly the same code is redeployed to the block chain. In fact, the 2,988,133 smart contract creations considered only contain 596,229 different smart contract implementations. From these, 45,107 implementations have been deployed more than once with exactly the same code. In Fig. 4 we show the number of smart contracts with multiple redeployments.

Due to the young age of the Ethereum blockchain the figures presented are heavily biased by recently created contracts that

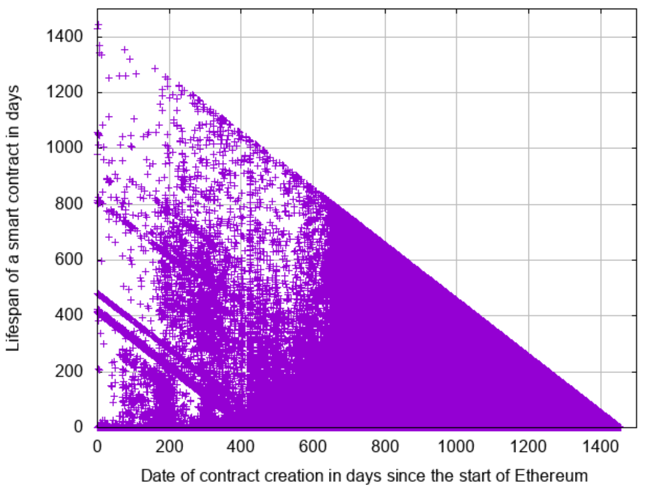

Fig. 5. Relation between date of contract creation and lifespan of the contract

can not yet be considered as long-living. This gets even worse due to the increase of contract creations per month as shown in Fig. 1. To get a feeling about the distribution of the contracts lifespan in relation to the date of their creation, we plotted the lifespan of contacts in relation to the date of their creation in Fig. 5. Early after the release of the Ethereum (at `x=0`) a few long-living and some only short living smart contracts have been created. For later creation dates more and more long smart contacts with higher lifespans are created. It is interesting that there are points of time where large amounts of short, medium, and long lifespans have been created. The two most obvious ones are after 200 and 600 days.

In summary, compared to the lifetime of the Ethereum blockchain, a significant amount of smart contracts deployed to the Ethereum blockchain can be considered as long-living within the Ethereum ecosystem. However, in relation to time frames traditionally considered for long-living software systems, it has to be observed if the future lifespan and usage patterns of smart contracts converge to the general understanding of long-living software systems. Regarding this, our findings indicate that smart-contracts might become long-living. For long-living smart contracts, we have to evaluate blockchain characteristics colliding with smart contract maintenance.

B. Update Patterns

If a blockchain is used for smart contract deployment and distribution, a smart contract can neither be updated nor deleted once it is deployed. However, most systems need to provide update functionality e. g. to deal with changing environments. This holds especially for long-living systems which, according to the previous subsection, smart contracts can be.

We discovered two main patterns used to update smart contracts on the Ethereum blockchain

a) Redeployment: The simplest update pattern is the creation of a new smart contract where all changes are applied. Here, we have to distinguish between smart contracts that are created for single-use, and smart contracts which can be used multiple times with a single deployment.

Deployment per use. In this case the recreation of the smart contract is part of the concept and allows to maintain the smart contracts code in between the single uses. While for smart contracts without changes a simple comparison of the byte code of an existing contract known to behave as expected is enough the updated contract has to be checked in detail. However, the same mechanism can be applied here as for a smart contract where no version known to be secure exists.

Single deployment. It is possible to implement smart contract with some kind of session management to enable the use for multiple parallel sessions. Once the address of the smart contract is publicly known, users can be certain to communicate with the same smart contract on each interaction. The redeployment of such a smart contract, which will technically result in a new smart contract with a new address, raises the issue of distributing the new address since the fixed address is part of the contract usage design.

One approach to tackle this issue are centralized places for the exchange of the current address for a specific smart contract. E.g. this can be smart contract market places, the website of the contract deployer, or a client software used for the execution of the smart contract. Each of these solution contains some kind of Single-Point-of-Failure.

To avoid the usage of the old deprecated smart contract the deployer has to revoke the old one. This step might not be executed by the deployer due to various reasons, like expecting too high cost for the self destruct or due of laziness. If this self destruct does not happen or is not possible, e.g. because required authentication data has been lost, users unaware of the new smart contract might continue using the old one containing bugs or security issues.

b) Proxy Smart Contracts: A pattern frequently used is to deploy a pair of smart contracts where one of the contracts functions as a proxy for the other smart contract. The proxy contract has a changeable variable containing the address of the actual smart contract doing the work. Whenever a newer smart contract version is released, the value of this variable is updated. If the proxy is called, it calls the newest version of the smart contract that has been released. While this pattern seems to behave very well in practice, it violates the assumption that every time the same address is called with the same input, the smart contract at this address behaves the same.

As there is currently no update mechanism that provides a trust-free redirect to the new smart contract, updates should be only performed for security critical changes and should be well documented.

C. Challenges and Future Work

Based on the previous discussions we identified challenges that should be tackled in the future. The overall challenge is how to tackle the maintenance problem of long-living smart contracts. To deal with this challenge, we identified the subchallenges present in the following.

While we have already been able to extract much information from the Ethereum blockchain there are still more things to be studied in detail.

At first, we have to provide a more precise definition of a long-living smart contract. In this work, we only discussed informally, based on quantitative data, if there exist long-living smart contracts on the Ethereum blockchain. A more formal definition would allow to explicitly select the long-living smart contracts and to study them in depth.

Second, we need more detailed information on the versions of long-living smart contracts. In this work, we searched for smart contracts that have been redeployed without changes. However, we expect many contracts to be redeployed with small changes, reflecting an iterative development process. The challenge is to detect and analyze them. I.e., we have to study which changes have been made between different versions of a smart contract. This can allow us to get a deeper understanding of the reasons for maintenance and allow us to reduce the amount of redeployments by developing best practices and tool support for reducing the probability for errors.

Extracting usage patterns from the smart contracts and studying their pros and cons is a first step for applying mature software engineering to smart contracts. While there is already some work on this issue, the challenge is still to get a deeper understanding of good patterns for smart contracts. Also these usage patterns should be compared to the usage patterns of traditional software with the aim of a knowledge transfer.

Another challenge is to search and identify more update patterns and to quantify their usage. While we already identified two update patterns there might be more that we are currently not aware of. The open question is if we will find always the same patterns to be used frequently or a larger variety of update patterns. Again, a deeper understanding on update patterns for smart contracts will increase the understanding on how smart contracts are deployed and developed. Furthermore, there are well-proven update patterns in traditional, non-blockchain software systems. It has to be checked if these patterns could be applied to smart contracts stored on a blockchain. Particular attention should be payed to software systems providing the possibility to be updated while being handle long-running transactions.

In general, one has to follow the Ethereum community discussions to learn in which direction smart contracts will evolve and which concepts should get part, e.g. of Ethereum, to allow maintenance of smart contracts. One solution to ease the maintenance of smart contracts could be to allow the specification of additional information when executing a selfdestruct, e.g. by adding a pointer which points to the succeeding version of the smart contract to be destructed.

IV. Related Works

Besides us also other researchers are studying the usage of smart contracts. In this section we discuss their works and relate them to our work.

Chen et al. analyze the Ethereum blockchain for occurrences of ponzi schemes [10]. For this purpose they utilize a pattern detection on the contracts available on the Ethereum block chain. A similar approach could be used by us to detect versions of contracts.

Bartoletti manually classified contracts published on the Ethereum blockchain into different categories [11]. Furthermore, they studied which patterns have been implemented in the contracts, e.g. an authentication.

Parizi et al. studied the effectiveness of open source security testing tools for smart contracts on ten contracts taken from the Ethereum blockchain [12]. Effective tools for detecting issues early and avoiding them can help to also avoid the issues arising from the need to update already published smart contracts discussed in this work.

While we discuss update patterns for smart contracts, Wohrer et al. are discussing security patterns in their work [13]. They are presenting a collection of patterns describing solutions to typical issues.

Liu et al. discuss design patterns for smart contracts and even identified two patterns categorized as creational patterns [14]. Unfortunately, those creational patterns do not consider maintenance and updates of the contract.

V. Conclusion

In this work we conducted an initial study of smart contracts deployed on the Ethereum blockchain and discussed whether smart contracts can or should be considered to be long-living software. We show, that there is a conflict between the idea of having immutable smart contract code and the need for updates as part of a software's life cycle.

Despite lacking a formal definition of long-living smart contracts, we discovered that a significant amount of smart contracts could be considered to be long-living. In general, long-living software needs the capability of updates, e.g. for adoption to unforeseen changes in the environment. This also applies for software in blockchain context, like smart contracts. For the smart contract code itself well known maintenance principles can be applied. However, we are currently not aware of e. g. refactoring approaches for smart contracts [15].

More challenging is the secure update of the deployed smart contracts. While proxy smart contracts seem to work very well in practice, it might raise the issue of smart contract address distribution for securely informing clients about an updated smart contract version. The key principle of an identical behavior per invocation might be violated by the use of proxy smart contracts. Unfortunately, for now we can not recommend a better solution if the contract should stay available under the same address. However, we see a high need for mechanisms to update the code of smart contracts. For this reason, we want to motivate discussions and research to address the conflict between reliable and and secure smart contract platforms and the need for updating the smart contract code.

As changes in long-living systems cannot always be prohibited and smart contracts can be long-living systems, smart contracts platforms should provide transparent ways to update the code of smart contracts and allow everyone interested in calling the contract to trace the changes.

In our future work we are going to get a deeper understanding on the versioning of smart contracts as well as in update mechanisms. Based on this understanding we are going to develop best practices and additional tool support.

References

[1] D. L. Parnas, "Software Aging," in *ICSE*. IEEE, 1994, pp. 279–287.

[2] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching," in *ASE*, Sep. 2016.

[3] European Parliament and Council of the European Uninon, "Regulation (EU) 2016/679 – General Data Protection Regulation (GDPR)," in *Official Journal of the European Union*, 2016.

[4] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Springer Science & Business Media, 2012.

[5] G. Wood *et al.*, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," *Ethereum Project Yellow Paper*, 2014.

[6] C. Cachin, "Architecture of the Hyperledger Blockchain Fabric," in *DCCL*, vol. 310, 2016.

[7] S. Nakamoto *et al.*, "Bitcoin: A Peer-to-peer Electronic Cash System," 2008.

[8] "Blockchair Database Dumps," online, https://blockchair.com/dumps, [accessed Feb 20].

[9] J. Bürger, "Recovering Security in Model-Based Software Engineering by Context-Driven Co-Evolution," Ph.D. dissertation, 2019.

[10] W. Chen, Z. Zheng, E. C. . Ngai, P. Zheng, and Y. Zhou, "Exploiting Blockchain Data to Detect Smart Ponzi Schemes on Ethereum," *IEEE Access*, vol. 7, pp. 37 575–37 586, 2019.

[11] M. Bartoletti and L. Pompianu, "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns," in *Financial Cryptography and Data Security*, 2017, pp. 494–509.

[12] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains," in *CASCON*, 2018, pp. 103–113.

[13] M. Wohrer and U. Zdun, "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity," in *IWBOSE*, 2018, pp. 2–8.

[14] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, *Applying Design Patterns in Smart Contracts*, Jun. 2018, pp. 92–106.

[15] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.