IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Byzantic: Privacy-Preserving Collateral Reduction for DeFi Protocols

Author:
Ioan-Daniel Savu

Supervisor:
Dominik Harz

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2020

Abstract

Decentralized Finance (DeFi) is an ecosystem that originated in the Ethereum blockchain, whose creation is a response to the 2008 Global Financial Crisis. It aims to provide an alternative to the traditional financial system by promising increased transparency, censorship resistance and better accessibility.

DeFi is still in its early stages, and its main offering is over-collateralised loans, the equivalent of mortgage-backed securities. At the moment, debt is issued assuming borrowers are self-interested, choosing to default if the escrowed assets become less valuable than the loan. Moreover, legal action cannot be taken against agents who ”misbehave” because identities in Ethereum are pseudonymous. As a result, borrowing in DeFi requires a collateral surplus to secure the lender against price volatility. A significant disadvantage of this protection measure is the opportunity cost of the locked-in collateral.

The current work presents Byzantic, a reputation system built on Ethereum, aimed at reducing the opportunity cost associated with over-collateralisation. Byzantic intermediates transactions between agents and DeFi lending protocols, measuring agent reputation based on the actions they perform. Because lending protocols differ from each other, the ideal reputation is subjectively defined by each protocol governance. Good reputation reduces the amount of excess collateral needed for borrowing to as little as zero, making the loan fully collateralised instead of over-collateralised. Liquidity is thus unlocked, reducing price volatility and improving security in DeFi. The discount in collateral ”good” agents gain is transferable across protocols that integrate with Byzantic up to 100%. In addition, Byzantic offers agents the possibility of building reputation while remaining anonymous, based on a new architectural design pattern proposed in this project.

We carried out a behavioural analysis of agents in Compound, Aave and Synthetix, three of the most popular DeFi protocols, during the first quarter of 2020. With the newfound insights, we simulated Byzantic user reputation over the same period. The 10% most active users, who originate more than 85% of transactions, achieve a Collateral Reduction (CR) of 30% on average. CR shrinks to 3% during the ”Black Thursday” Ethereum price crash in March, confirming the resilience of the system against sharp economic downturns. If Byzantic were to achieve 30% CR across all of its target protocols, it would create 1.5B USD of additional liquidity as of August 2020.

Acknowledgments

First, I wish to thank my supervisor, Dominik Harz, for his tremendous support and encouragement throughout the last five months. His enthusiasm made this project a very rewarding experience, in spite of only meeting once in person.

I also had the pleasure to collaborate with Tom Waite of Aztec Protocol, who was always willing to help me on topics related to his work. I would also like to thank Lewis Gudgeon for his guidance on the evaluation of this thesis and Professor William J. Knottenbelt for advising me as a second marker.

Lastly, I want to express my gratitude to my family and close friends for their love and invaluable support. Without them, this work would not have been possible.

Contents

Chapter 1

Introduction

1.1 Motivation

The offerings of the traditional financial system range from lending and savings accounts to exchange markets where securities can be traded, wealth management and insurance [1]. However, after the 2007-2008 financial crisis, trust in this system was severely undermined [2]. The literature on the topic agrees that secrecy surrounding financial risk models was one of the main factors that led to the crisis [3]. The intellectual property aspect of risk models for credit scoring protected financial companies from having these models audited. This lack of transparency, coupled with the trust financial institutions enjoyed, resulted in misusing the risk models to overvalue low-quality assets when issuing debt.

Had risk models been completely transparent, the crisis might have been avoided. This is the main principle behind DeFi, which aims to provide the same services as the traditional financial system, but in a fully transparent way. Moreover, in DeFi, authoritarian governments should be unable to restrict certain individuals from participating in the market. Lastly, even people in developing countries, without access to sophisticated traditional financial opportunities, should be able to participate in the global economy - all they need is internet connectivity and a computing device.

DeFi is built on technology that launched only five years ago: the Ethereum blockchain. As such, it still faces scalability-related challenges and liquidity shortages, which prevent widespread adoption. While the former is on track to being solved by the next version of Ethereum [4], low liquidity remains a critical problem to DeFi stability [5].

When agents of an economic system trust each other, they benefit from increased utility (Section 2.6). Reputation is an objective measurement of trust, so systems which measure user reputation also benefit from increased economic utility. In the context of lending, borrower reputation has been named ”intangible collateral” [6], because collateral and reputation can act as substitutes to each other. As collateral depreciates, the borrower is tempted to default on their loan. However, the prospect of being able to take loans on better terms in the future can counteract such default risk. Thus, if economic agents are encouraged to build reputation as good borrowers, the reputation framework acts as insurance against default risk in case of collateral devaluation [7]. Whereas credit scoring is used ubiquitously in traditional finance to rate and reward borrowers, reputation is not even being measured in DeFi at the moment.

1.2 Proposed Solution

We propose Byzantic, a novel Ethereum protocol that measures reputation not only for borrowers, but for all economic agents participating in lending services in Decentralized Finance (DeFi). While credit scoring is an objective metric, Byzantic is a versatile tool whereby protocols that integrate with it configure what constitutes good reputation and how much of the reputation in other protocols is transferable to their use case. There is no formal notion of reputation in DeFi as of yet, and as such all interactions happen under the assumption that the other parties are dishonest. A consequence of this assumption is the usage of over-collateralisation to issue debt, which protects lenders against the devaluation of security deposits. Our solution aims to bridge this gap, by providing DeFi users with the opportunity to build reputation, the intangible asset that can replace physical security deposits.

Byzantic can reduce collateral requirements from over-collateralisation (e.g. collateral amounts to 150% of the borrowed amount) to as low as full collateralisation, where the value of collateral is equal to that of the loan. By doing so, Byzantic unlocks assets that were previously locked-in, adding liquidity to DeFi. Furthermore, Byzantic integrates with DejaVu, a new design pattern that improves privacy when reputation is being tracked.

Byzantic can be applied to any protocol involving some form of lending. In DeFi, this includes even Synthetix, a protocol used for minting derivatives, the issuing of which requires a 750% collateral rate. Even if it is not directly a lending protocol, Synthetix works as one, because its derivatives are issued with interest that needs to be repaid when the derivatives are ”burned” [8]. Besides Synthetix, Byzantic can integrate with: Aave, Maker, yearn.finance, Compound, InstaDApp, dYdX, ForTube, DDEX, RAY, Dharma, bZx, which in total account for 77% of locked-in liquidity in DeFi. Given that Byzantic can free up collateral, its impact becomes more significant as more value flows into DeFi (Figure 1.1).

In addition to reducing collateral surplus requirements, the reputation Byzantic builds could be used for other purposes too. For instance, agents with high reputation might trust each other enough to pool their investments and share the risk, without fearing that one agent will run away with all the pooled funds. Another example is using Byzantic reputation as a credit scoring mechanism, allowing for under-collateralised debt.

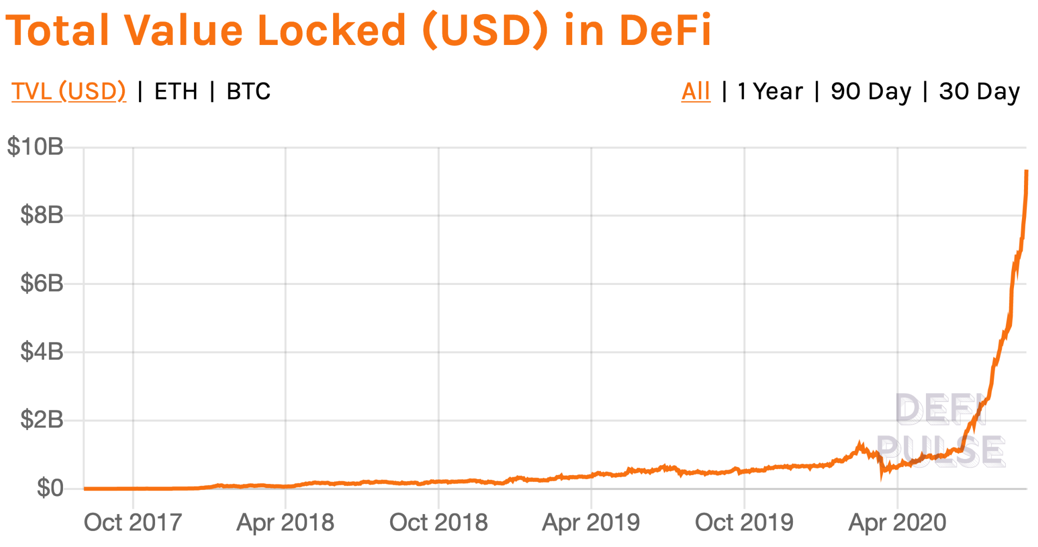


Figure 1.1: Evolution of DeFi locked-in collateral [9]. The market crash in March 2020 is most likely due to the COVID-19 pandemic, since it coincided with a sharp stock market decline [10]. It has been called ”Black Thursday”.

In this report, the terms *user* and *agent* are used interchangeably.

1.3 Objectives

First, the current project aims to **reduce collateral surplus as much as possible**, with minimal increase to default risk, even during price drops. Second, even if reputation-based collateral reduction has been shown to be secure [11], DeFi protocols still need to be convinced to trust and integrate with a reputation system. **Integrating with Byzantic should require minimal changes to existing smart contracts**, such that the risk of introducing a security vulnerability is diminished. Third, the smart contracts that constitute Byzantic should prove **resiliency against attacks** such as double-spending, transaction reordering and Sybil identities. Fourth, agents who use Byzantic should have the option of **preserving anonymity while building reputation**.

1.4 Contributions

The current work brings the following contributions.

- **Design of a reputation system for DeFi**. We propose Byzantic, a novel framework that can be applied to any decentralized ledger to measure DeFi reputation. We examine the use of this system in safely reducing over-collateralisation requirements to full collateralisation (100% of the borrowed amount), improving agent utility and adding liquidity to DeFi. Each protocol that integrates with Byzantic decides what type of user behaviour is rewarded and how data from other protocols is aggregated to compute reputation. Byzantic can also be configured to reflect how much collateral can be lowered, how soon users can achieve good reputation and how quickly changes in user behaviour should determine the re-evaluation of reputation. Because changes to the configuration of a protocol may skew aggregated reputation in other protocols, it is possible to distrust recently updated protocols by default, excluding their data from the aggregation. This framework is presented in Chapter 3. Byzantic is an extension to Balance [11], which introduced a thorough game-theoretic analysis of why collateral can be safely reduced in single protocols using reputation.

- **Solidity implementation of Byzantic**. Chapter 4 documents the design of the implemented system. After iterating over the system architecture three times (Chapter 5), we believe the final implementation offers the best usability-security trade-off given the system requirements. To encourage integration with Byzantic, we also set up a documentation website that automatically updates itself based on smart contract comments (Chapter 11).

- **A new design pattern for preserving privacy**. In order to protect user privacy while tracking reputation in the Ethereum public ledger, we built and integrated with DejaVu, a novel schema that provides k-anonymity guarantees. It uses zero-knowledge proofs, a blockchain-layer relay and a transaction mixing service. Besides anonymously tracking reputation, DejaVu can be used for privacy-preserving joint accounts, such as organisation accounts where employee privacy is preserved while the business of the company is transparent. Chapter 6 details the integration of Byzantic with DejaVu.

- **Economic stress-testing of Byzantic using behavioural data**. In Chapter 8 we present the first user behaviour analysis for Compound, Aave and Synthetix, three of the most popular protocols in DeFi, with a particular focus on the ”Black Thursday” Ethereum price crash. The results provide a better understanding of what constitutes good user reputation. They also show that user behaviour after the crash seems to oppose findings from market cycle psychology [12], possibly because of the fear of missing out. Using this data, we designed a custom economic stress-testing package written in Python, which protocol governances can use to tune their system configuration. We have exemplified this process in Chapter 9.

- **Security testing with smart contract audit tools**. We used two static analysers that are complementary in identifying vulnerabilities, such as transaction reordering and reentrancy (Chapter 10). Moreover, a generic lending protocol was implemented (Chapter 7) to perform integration testing to a high level of coverage.

1.5 Limitations

Byzantic has the following drawbacks.

1. **Increased transaction costs**. Byzantic runs additional instructions in the Ethereum Virtual Machine when it intermediates transactions, which cost an additional 73000 gas than a direct transaction - a 247% increase in our tests. If the DejaVu pattern for privacy is used, the same call costs 381000 more gas than a direct call - an increase of 867% (see Section 10.8).

2. **Only benefits the most active users**. Byzantic consistently reduces collateral only for the most active 10% of protocol users, who perform at least 85% of all actions in Compound, Aave and Synthetix. The rest of the users are most likely acting too infrequently to benefit from Byzantic. However, an exact analysis comparing gas costs and collateral reductions for these users has yet to be performed.

3. **Anonymity is not guaranteed**. The DejaVu design pattern provides the k-anonymity inherent its underlying components: a blockchain-layer relay and a transaction mixer. This means that the more users with the same transaction patterns, the better the privacy someone can achieve. This is similar to how the Tor network cannot guarantee privacy.

4. **Difficult setup**. The high number of parameters makes configuring Byzantic an error-prone process. This thesis presents guidelines and a stress-testing simulation tool in an attempt to ameliorate this issue.

5. **May require protocol redesign**. If usage of our solution is not considered when a protocol is designed, it might be too late to simply ”add” Byzantic. Collateral checks can occur in multiple places and may not even be refactored into a single component, so integrating with Byzantic may need to be delayed until the next major release of a protocol.

6. **External attacks**. Agents can still interact with lending protocols outside of Byzantic, and those actions cannot be tracked. Our analysis shows desired and undesired actions to always be complementary, such as borrowing being desired and loan repayment being undesired (Section 9). With the scoring configurations we propose, protocols need to prevent users from repaying loans on behalf of Byzantic identities. This is so that outside behaviour cannot manipulate reputation. However, if a scoring configuration is found which rewards undesired actions with zero rather than a negative score, then outside activity is no longer a threat.

7. **Fixed action rewards**. An implementation detail that can be improved is replacing fixed action rewards with a function that rewards users based on transacted amount. We considered this simplification good enough because of the high transaction fees in DeFi. They discourage users from performing the same action multiple times with small values, encouraging single transactions instead with the ”full” asset amount.

1.6 Ethical Considerations

The current project involves the collection and processing of all transactions between users and protocols that integrate with Byzantic. Such transactions are publicly available on the Ethereum blockchain, but tracking can be prevented by using more than one account. This prevents an accurate user profile from being built. Byzantic uses a unique address for every identity, but this thesis shows how integrating with DejaVu can protect privacy.

Names or other personal details are not included in transaction concerning Byzantic, but user profiles may be built to track financial behaviour, infer socioeconomic class or other aspects of personal nature. The publicly available transaction data could be misused for malevolent purposes. The risk aversion of large segments of population could be extrapolated by correlating changes in average transacted value with the period of the year or political context. Grasping the risk appetite of population segments can serve for more intrusive marketing and targeting of political messages. However, these risks are not particular to Byzantic - they are risks inherent to participating in any activity on the Ethereum blockchain.

1.7 Legal Considerations

Byzantic is open-source, as is every third-party software it uses. It is thus free for commercial use.

Analyses have shown that criminals are increasingly using cryptocurrencies for illegal purposes due to their privacy-preserving properties and the usability convenience compared to cash [13]. The use case of Byzantic, however, has nothing to do with peer-to-peer payments. The project aims to improve decentralized lending through a reputation system. Purchasing illegal goods is only facilitated by Byzantic to the extent that protocols that integrate with it facilitate this. It can help terrorist organisations as much as it can help counter-terrorist organisations.

Chapter 2

Background

This chapter introduces the concepts Byzantic relies on. It uses blockchain (Section 2.1) as a public immutable data store, both for transactions and source code. By doing so, Byzantic is completely transparent and open to inspection at all times. Byzantic is a decentralized financial service called a Dapp, operating on the Ethereum blockchain (Section 2.2). Ethereum identities are pseudonymous, and Byzantic offers users the possibility of building reputation without compromising privacy (Section 2.3). Byzantic was built on Ethereum because it needs to interact with all the other Dapps that form the DeFi ecosystem (Section 2.4), which are themselves built on this blockchain (Section 2.5). It aims to measure agent reputation across Dapps (Section 2.6) to lower the over-collateralisation rate used in trustless lending. Importantly, Byzantic reduces collateral without compromising protocol (i.e. Dapp) security (Section 2.7). All of this is possible because Ethereum facilitates the creation of custom financial agreements via smart contracts (Section 2.8).

2.1 Blockchain

Fundamentally, a blockchain is a decentralized, distributed, append-only store of data. There is no central authority or trusted node in the blockchain network, which means peers organise themselves at the expense of added complexity [14]. The main difference between blockchain and other distributed data storage systems stems from its decentralized nature [15]. It allows network peers who do not trust each other to work collaboratively towards ensuring the immutability of the data [16]. New records are added as part of a fixed-size block, which is appended to the previous blocks using cryptography (see Figure 2.1). A block is replicated across the entire network and its validity is verified by every peer [17].

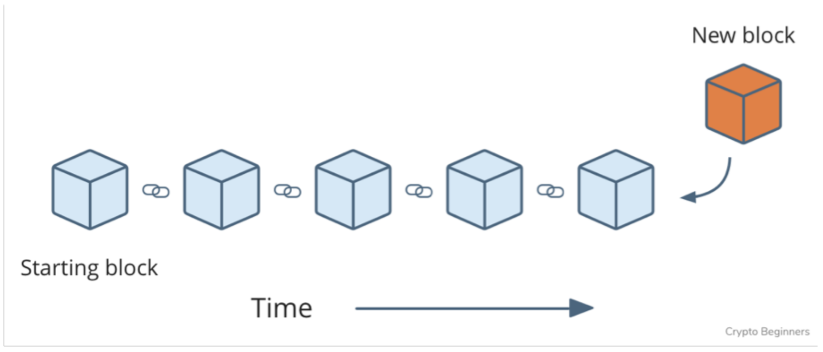


Figure 2.1: Adding a block to the blockchain. Credits to Colorado State University [18].

2.1.1 Bitcoin

Blockchain technology was introduced in 2008 under the pseudonym of Satoshi Nakamoto [17]. Bitcoin is an electronic payment mechanism that relies entirely on digitally stored value - the first one of its kind to reach widespread adoption [19]. The type of financial asset it represents has been labelled ”cryptocurrency”, for its use of cryptography. Because Bitcoin is a financial asset, the records comprising the blocks in the chain are financial transactions.

2.1.2 Liveness, Consensus and Proof of Work

Distributed systems have a liveness property which mandates that something good must happen eventually [20]. In the case of blockchain, this is achieved by all peers agreeing on which block of data to add next to the chain. In other words, peers must reach consensus for the system to be functional. The consensus algorithm most commonly used in blockchain systems is Proof of Work (PoW) [21], whose participants are called miners.
A PoW algorithm requires miners to solve a computationally expensive search puzzle: finding an input that when applied to a cryptographic hash function produces a hash belonging to a target set (e.g. a hash with three zeroes in the beginning). A key feature of this class of algorithms is the asymmetry between generation and verification. The difficulty of finding a solution is ensured by the ”puzzle-friendly” property of the hash function [19]: there is no approach to finding a solution that is faster, on average, than a brute-force approach.
The exact moment when PoW is applied is when a miner bundles transactions together in a block, because the network will only accept blocks that satisfy the PoW. But why would miners be willing to expend their computational resources and electricity to help the blockchain maintain its liveness? The answer is the coinbase transaction, which occurs once per block and has a predetermined value, which rewards the miner. The coinbase transaction is how new coins are minted, as it only has a destination but no sender.
Thus, the more difficult the puzzle, the longer miners will take before a solution is found. This fact is critical in the functioning of blockchains because the puzzle difficulty impacts the throughput of the system: if blocks are found quickly, then

transactions are bundled quickly into blocks. But when blocks are generated often, the risk of a double-spending transaction (one which uses the same funds twice) increases. If two blocks have conflicting transactions (e.g. double-spending ones), only one will be accepted by the peer-to-peer network (whichever one propagates faster). However, it has been shown that low mining difficulty results in less secure blockchains [22].

2.1.3 Consistency, Forks and Double-Spending

In addition to liveness, distributed ledger technologies like blockchain must also satisfy a consistency property, where every node in the distributed system should see the same state at all time. The Bitcoin and Ethereum blockchains run over a network with message delays (the internet), and are subject to the limitations explained by the CAP theorem: when a network partition occurs (P), a distributed system can achieve either high availability (A) or strong consistency (C), but not both. The Bitcoin and Ethereum blockchain follow an AP model, opting for high availability and only eventual consistency. Pass et al. (2017) [23] formally define the consistency of such blockchains as T-consistency: nodes agree on the current chain except for a small number of blocks - T - that were recently appended and have yet to fully propagate through the network, assuming there is a bound on the network delay. In a network where delays are unbounded, however, neither consistency nor consensus 2.1.2 can be achieved.
Because the blockchains in discussion only have eventual consistency, there may be more than one candidate new block at a given time. Miners might begin mining on top of any of them if they are valid. Such a situation is called a fork, and in PoW, the main chain is the one with the most work. As long as all branches of a fork have the same amount of work, there is no global main chain, which can happen during a network partition. Forks threaten the security of a blockchain because they mean that transactions are never final. They become ”more” final the more blocks are appended after their block. Naïve users, who do not wait for transactions to settle (i.e. do not wait for the block containing their transaction to become part of the T-consistent section of the ledger), are easy victims to double-spending attacks. It has been shown that in Ethereum, 37 block confirmations, or about 10 minutes, are enough to prevent double-spending even when adversaries control 30% of the hash power (mining power) in the network [24].

2.2 Ethereum

Bitcoin proved that Blockchain is a reasonably robust tool for storing value through the internet [25], an inherently insecure piece of infrastructure. The blockchain does allow for the creation of custom transactions using Bitcoin scripts, and there is even a language for modelling smart contracts built on top of the Bitcoin scripts [26]. However, these features are not as expressive as the opcodes offered by the Ethereum Virtual Machine (Section 2.2.1). Ethereum is a Turing-complete state machine [27], in which transactions are state changes. This means that, as opposed to Bitcoin, Ethereum is able to perform any sort of financial agreement, the only limiting factor being gas costs. As blocks have a fixed gas limit [28], if a transaction exceeds that cost it will not be ”computable” by the Ethereum Virtual Machine.

2.2.1 The Ethereum Virtual Machine

In Ethereum, financial agreements are defined by Smart Contracts, or computer code usually written in Solidity and compiled to EVM bytecode. This bytecode is executed by a global network of computers which comprise the EVM - a stack-based virtual machine, which supports both volatile and persistent storage [29]. To prevent Denial-of-Service attacks, storage and ”CPU time” in the EVM have a cost associated to them, expressed in Gas [30]. The amount of gas required to run a transaction is measured using the fixed gas cost of each EVM bytecode instruction. The gas is paid to the miner, so transactions can have their gas costs ”artificially” inflated in order to incentivize miners to include them into the blockchain faster.

2.2.2 Contracts and Accounts

The EVM bytecode of a Smart Contract is deployed onto the blockchain by issuing a transaction with an empty destination address. The state transition creates a contract account (i.e. an address) with the associated bytecode, whose methods can be ”called” by issuing transactions with the new contract’s address as the recipient.

Both Ethereum users and contracts have an account (a public key used as an address) associated to them. This way, the EVM can keep track of account balances by mapping addresses to amounts of Ether, the currency of Ethereum. Ether is used to pay for Gas fees, but also as a store of value. The account-based approach to storing state is in contrast with Bitcoin’s ”Unspent Transaction Output”-based approach (UTXO) [31]. In Bitcoin, the balance of an address is not stored explicitly anywhere. Instead, it is calculated as the sum of all unspent outputs from transactions that have occurred. The account-based model is better suited for smart contract development, since checking an account’s balance in the UTXO model would require computation (aggregation) to be performed every time. The same reason makes validating transactions is simpler - no need to check all the UTXOs. The drawback of Ethereum’s approach of explicitly storing account balances in every block is that it takes up more storage space.

2.2.3 Communication Protocols

Ethereum is a peer-to-peer overlay network that operates over the internet. To achieve liveness and consensus (Section 2.1.2), its nodes must be able to communicate. To communicate, a new node must first discover some nodes in the network and connect to them.

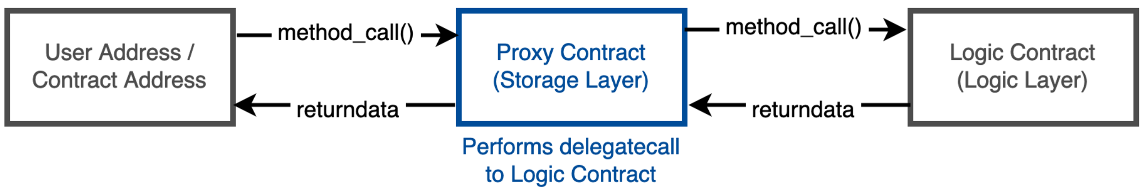

User Address / Contract Address
method_call()
returndata
Proxy Contract (Storage Layer)
Performs delegatecall to Logic Contract
method_call()
returndata
Logic Contract (Logic Layer)

Figure 2.2: Standard Upgradeability Proxy Pattern

Node Discovery

Ethereum uses a Distributed Hash Table (DHT) to store information about nodes. It is a modified version of the Kademlia DHT [32] and is used because it yields a low-diameter network topology that supports fast lookups. Data is replicated across every node in Ethereum, so the DHT is only used for node discovery and efficient routing. This is in contrast with the classic Kademlia implementation, which also supports index-based data storage and retrieval [33].

The node IDs used in the DHT are the Keccak-256 hashes of the node addresses (which are ECDSA public keys). The DHT partitions the network according to the distance from the current node ID. That distance is $\lfloor log_2(a \oplus b) \rfloor$, and produces 257 partitions. To confirm a node is live and connect with it, two UDP messages, `PING` and `PONG` are sent as a request-reply sequence. Then, to query that node for a path to a target node, another pair of UDP reply-request messages is used: `FINDNODE` (with the target node as a parameter) and `NEIGHBORS`. `NEIGHBORS` returns the 16 closest nodes to the target node in its routing table. Based on the two pairs of messages, a recursive lookup is performed until the target node is found.

RLPx Transport Protocol

RLPx is a TCP-based protocol that forms the transport layer of Ethereum communication. Following an initial handshake to establish connection, messages are exchanged in encrypted form. The most important protocols built on top of the RLPx transport layer are the Ethereum Wire Protocol (*eth*), the Light Ethereum Subprotocol (*les*) and the Parity Light Protocol (*pip*) [34]. Of these three, *eth* is the most notable, as it enables chain syncing, block propagation and transaction exchange.

2.2.4 Upgradeability Proxy Design Pattern

One of the fundamental aspects that enable blockchain technology to serve as the backbone of a new financial system is the immutability of state stored in it. No individual can change smart contract state or transaction details. However, the immutability of blockchain is also a disadvantage in the case of upgradeability. Once deployed, the code of a smart contract cannot be changed to release bug-fixes or new features. This is where the proxy design pattern comes in useful, which uses the `delegatecall` EVM opcode to call the code of another contract (Logic Contract in Figure 2.2) and execute it in the context of the caller contract (Proxy Contract in Figure 2.2) [35]. This way, `msg.sender` and `msg.value` are preserved. Using `delegatecall` also means that all the state changes made by the called contract are registered in the caller contract. Thus, the caller contract can be considered the storage layer, and the called contract can be considered the logic layer. This pattern allows programmers to upgrade smart contracts by deploying a new Logic Contract and then changing the `delegatecall` target address in the Proxy Contract. No state is lost during the upgrade, because the Proxy Contract is the one tracking state. See Figure 2.2 for a visual representation of this technique.

2.3 Privacy

Ethereum is a permissionless blockchain, meaning that anyone can participate without asking for permission from any entity. An advantage of this property is that there are no ”hard” identities based on legal documents. An individual may possess more than one account in Ethereum, and, as long as the accounts never transact with each other, they are unlinkable.

Nonetheless, the fact that the Ethereum blockchain is completely open to public scrutiny raises serious concerns about privacy and may prevent widespread adoption. Two blockchain-layer measures have proven helpful in addressing this issue: Zero-Knowledge Proofs (Section 2.3.1) and Mixers (Section 2.3.2). These can be coupled with an internet layer solution, relay networks (Section 2.3.3), to further enhance privacy.

2.3.1 Zero-Knowledge Proofs

A Zero-Knowledge Proof (ZKP) is a way to prove knowledge of some information without revealing the information itself. An example is proving that a blue pen and a red pen have different colours to someone who cannot distinguish any two colours from each other. The colour-blind person holds one pen in each hand and both hands behind their back. Thus, the prover cannot see the pens. The colour-blind person can switch the pens from one hand to the other or leave them in their current position, and then shows the pens to the prover. If the prover repeatedly guesses correctly whether the pens have switched positions or not, they probabilistically prove that the pens indeed have different colours. In this scenario, ”probabilistically” means the proof is not guaranteed to be correct, since there is always the possibility of guessing whether the pens have been switched. The probability of this proof being wrong can be reduced by repeating the experiment. When the proof is verified as valid, the verifier (the colour-blind person) has not learned what the colours of the two pens are, thus having checked the proof in ”zero knowledge”.

ZK-SNARKs

Zk-SNARKs are the most widespread form of zero-knowledge proofs. The acronym means ”Zero-knowledge Succinct Non-interactive ARguments of Knowledge”. They enable the verifier to check that a computation is correct without actually running it. The properties of ZK-SNARKs make them an excellent solution in distributed ledger settings:

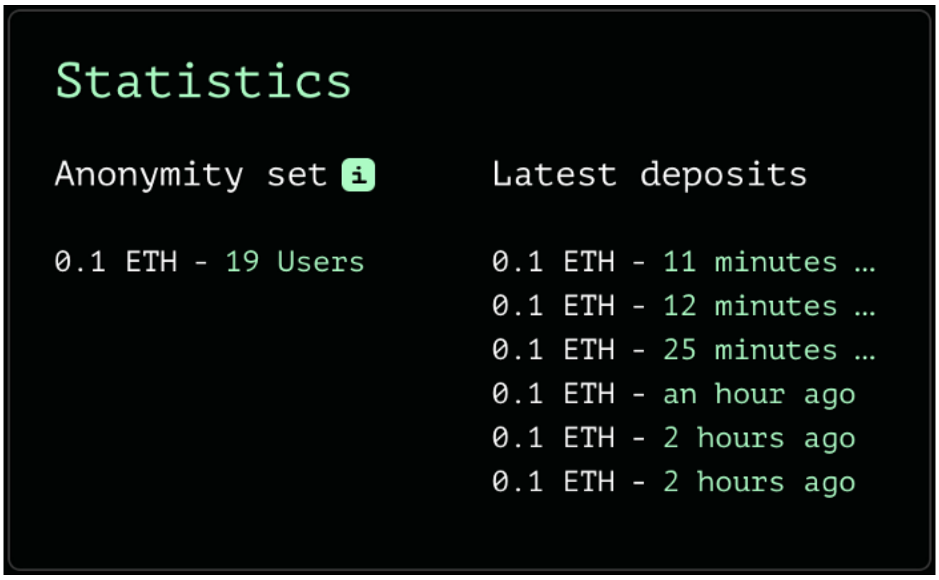
- *Zero-knowledge*. The prover does not learn the private parameters of the computation it checks.
- *Succinct*. The Ethereum platform is a state machine replicated on every full node. Its state changes are verified by every such node, which means that complex computations are very expensive. The succinctness property of ZK-SNARKs ensures that proof verification is relatively inexpensive. This property is achievable because the proof is only verified by the random sampling of one point, similar to running one experiment of the coloured pens proof in Section 2.3.1.

  ZK-SNARK proofs are expressed using the following mathematical relation, where A, B, C, H, Z are polynomial functions ”encoding” the proof:

  $$A(x) \times B(x) - C(x) = H(x) \times Z(x)$$

  Polynomial transformations are very computationally expensive, and performing them on-chain would severely limit the viability of ZK-SNARKs. Fortunately, picking a random x and checking if the equality still holds offers very good statistical guarantees too.
- *Non-interactive*. Validating ZK-SNARK proofs only requires the prover to interact once with the verifier: when sending the proof. This is also a great fit for the use case of Ethereum, because user addresses cannot ”react” in response to transactions sent to them, as opposed to contract addresses (Section 2.2.2). Non-interactiveness thus contributes to user experience, as users do not need to deploy a contract to help interactively validate the proof they are submitting.
- *Arguments of knowledge*. An argument of knowledge is different from a proof, in that the former must satisfy the additional condition of being verifiable by a computationally bounded validator (in polynomial time complexity). The latter can be shown valid by any computationally unbounded validator.

Besides the aforementioned properties, ZK-SNARKs also make heavy use of another concept: homomorphic encryption. This concept is used for evaluating the random sample x without knowing its value. For instance, an encryption E is multiplicatively homomorphic if, for any $a, b \in \mathbb{Z}$, the following equality holds: $E(a) \times E(b) = E(a \times b)$. Similarly, the polynomials mentioned in the ”succinctness” bullet point above can be applied directly to the encrypted value of x. Homomorphic encryption preserves the privacy of the private arguments of the proof.



Statistics

Anonymity set

Latest deposits

```
0.1 ETH - 19 Users        0.1 ETH - 11 minutes ...
                          0.1 ETH - 12 minutes ...
                          0.1 ETH - 25 minutes ...
                          0.1 ETH - an hour ago
                          0.1 ETH - 2 hours ago
                          0.1 ETH - 2 hours ago
```

Figure 2.3: Tornado cash k-anonymity set, with k = 19. The quasi-identifiers of this set are the asset (ETH) and the quantity (0.1).

2.3.2 Mixers and k-anonymity

Mixers provide anonymity by pooling together multiple transactions and then mixing them in such a way that makes it difficult to know which address sent which transaction. An example is Tornado Cash, which enables peer-to-peer anonymous transactions of value (Ether and ERC-20 tokens). First, the sender deposits funds in Tornado Cash [36] and secures them with a hash value. Then, the receiver must provide a zero-knowledge proof that shows they know the hash pre-image to redeem the funds. The receiver needs to wait for long enough, such that the **transacted asset** and **quantity** have been deposited by other addresses as well. If enough identical deposits have been made, the transaction is anonymised by those identical transactions. The **transacted asset** and **quantity** are called *quasi-identifiers* - taken individually, they may not uniquely identify a sender and a receiver; but together, they could. This type of anonymity is known as *k-anonymity* [37], because the transaction is ”hidden” amongst $k-1$ other identical transactions. Thus, a k-anonymity set is determined by its quasi-identifiers. The larger the value of k, the stronger the anonymity guarantees. A Tornado Cash k-anonymity example can be observed in Figure 2.3.

2.3.3 Relay Networks and Tor

In communications network topology, a relay is a node that intermediates a source and a destination [38]. If network communication is encrypted, external observers only learn that the sender communicates with the relay node and the relay node communicates with the receiver. If the relay is used by k senders for the same destination, it provides k-anonymity against network observers. There is no encryption in Ethereum, so the IP address of nodes broadcasting transactions is public [39].

However, if the publisher is a relayer which communicates with senders over an encrypted protocol, the sender still benefits from k-anonymity. If a single relay is used, the sender needs to trust it not to expose their identity. However, if a network of relays is used, similar to Tor, this issue is solved.
The Tor network itself can only be used if node discovery is performed manually. Tor is incompatible with the UDP-based node discovery protocol of Ethereum (Section 2.2.3) because Tor only supports TCP-based communication. The rest of communication is done over TCP (Section 2.2.3), where Tor could be used for protecting privacy.

Tor

Tor is an acronym for ”The Onion Router”. An *onion* is a multi-layer encrypted message, each encryption layer being analogous to the layer of an onion. As an example of onion routing, let us consider two relays (R_1, R_2) between two network hosts: Alice and Bob (an Ethereum node). The message sent by Alice to Bob would follow the path: Alice-R_1-R_2-Bob. If Alice establishes a different symmetric encryption session with each relay, she can send to R_1 a message of the form:

$$K_{R_1}(R_2, K_{R_2}(Bob, M))$$

where $K(D, M)$ represents data encrypted with encrypted with the shared key K, that once decrypted presents message M, that needs to be submitted to destination D. Because R_1 has the shared key K_{R_1}, it can decrypt the data it received, to learn the message it needs to send to R_2:

$$K_{R_2}(Bob, M)$$

After R_2 decrypts this message, it send M to Bob. If enough relays are used, traffic from a sender to a receiver becomes k-anonymised at each relay. Moreover, the final relay, which in the case of Bob being an Ethereum node obtains the encrypted transaction message, cannot know the IP address of Alice, because the message she sent became mixed with others in the relays. The relays become especially effective at hiding traffic if there is an entire mesh of relays available, of which Alice randomly chooses a path to Bob.

A relay network such as Tor does not guarantee anonymity, as a network observer analysing message latency could correlate messages sent by Alice to replies from Bob. Only if many other users are also communicating with Bob does Alice truly become ”hidden in the crowd”.

2.4 Decentralized Finance (DeFi)

In spite of its sophistication, the traditional financial system is based on a handful of institutions where power is concentrated and thus are susceptible to corruption and human error. Besides, because they are for-profit organisations, they need to add a significant profit margin to the price of their services in order to stay operational. Their offerings range from lending to savings accounts, exchange markets where securities can be traded, wealth management and insurance [1].

Traditional financial services can be replaced by decentralized counterparts on Ethereum, which together form DeFi. They are more transparent, easily auditable and provide more competitive prices or interest rates because there are no middle-men involved [40].

DeFi also has the advantage of accessibility. At the moment anyone on the globe can participate in DeFi if they have access to the internet, although this may change with new regulations or censorship. Nonetheless, no documents or lengthy verification processes are needed. A World Bank analysis has shown that of the 1.7 billion people without a bank account, two-thirds have access to mobile phones [41], so they could potentially take advantage of the financial services on DeFi.

2.4.1 Dapps and Custody

The financial services in DeFi are called Decentralized Applications (Dapps) and can be built by anyone with an Ethereum account. They are usually non-custodial [42], meaning that they do not store user funds, as opposed to banks. Custody of funds is a double-edged sword. On the one hand, account holders are in full control of their funds, so they are not affected if a Dapp gets hacked. On the other hand, if they lose their private key or it is compromised, nobody can help them recover their funds. With commercial banks, customer funds are protected by the central bank, at least in part [43].

2.4.2 Trust and Collateral

DeFi is a trustless environment for accessing financial products. There is no formal identity required, which means that misbehaving agents are outside any jurisdiction and cannot be punished by law. In effect, DeFi is being built expecting agents to be at best semi-honest and at worst Byzantine [44]. Instead of a legal system, the main protection measures are game-theoretical rules [45] which incentivize economically rational agents to comply with Dapp requirements. The most common such rule is over-collateralisation: to take a loan, agents must provide more value as collateral than what they are loaning. This way, it is more profitable for agents to repay their loan than to default. The over-collateralisation model also accounts for price fluctuations: if the ratio of collateral-to-loan-value is high enough, it is unlikely that the loan will become more valuable than the collateral. Overall, this solution is arguably worse than loaning in traditional financial systems, because the borrower needs to already have the money being loaned and a loan in DeFi simply provides more exposure to certain assets. In addition, there is an opportunity cost associated with the locked-in collateral. The advantage, though, is privacy, as a legal identity is not required.

2.4.3 Main Applications

Stablecoins

Stablecoins are of critical importance to a very volatile DeFi. Without a stable store of value, investors would be deterred from lending, for instance, because price volatility may outweigh their accrued interest. Stablecoins are pegged to stable stores of value, most commonly the US Dollar.

Lending and Borrowing

DeFi removes the need for a good credit score and usually provides better interest rates than banks [40], but loans need to be over-collateralised.

Asset Tokenization

Tokenization allows for greater market liquidity, fundraisers and unconventional securities. An expensive piece of art can be represented digitally as fungible Ethereum tokens which are similar to shares in a company and can be traded at finer granularity. There have also been cases where people used tokens as a decentralised form of fundraising, such as fashion house Saint Fame's $FAME Genesis Shirt [46]. More recently, people have even been tokenizing themselves [47].

Decentralized Exchanges

To use a classic exchange for trading, a user first needs to make a deposit on the platform, then trade, then withdraw their funds. Deposits and withdrawals not only incur fees usually, but they also mean that traders need to trust that the platform will not get hacked. Decentralized exchanges ensure their users can trade without temporarily losing custody of their securities.

Payments

An innovation DeFi brought about is trustless, streamed payments, where money is being paid continuously, like water from a tap. This can be used for pay-as-you-go services or for streaming salaries instead of paying them once per month [48].

2.5 DeFi Protocols

2.5.1 Balance

Section 2.4.2 explained how over-collaterlization is used as a solution to the lack of strong identities in Ethereum. It also highlighted the main drawback of this measure: opportunity costs.

Balance is a game-theoretic mechanism that explicitly models the utility agents gain from participating in a DeFi protocol [11]. When agents continuously perform desired actions, their collateral ratio is reduced without compromising protocol security. Assuming agents are economically motivated, Balance enables them to fulfil their goals (reducing opportunity cost) if they perform the actions that are in their best interest - a property called *incentive compatibility*. Not only do agents only have to act selfishly, but doing so increases the social welfare of the protocol. The consequence of using Balance in a protocol is that Byzantine agents are worse off trying to single-shot their reputation than if they acted truthfully to their intentions from the beginning. Moreover, by reducing collateral, Balance increases the ”economic bandwidth” of DeFi. Balance was designed to only be used in a single protocol, however, Byzantic takes it one step further by using it in a web-of-trust setting.

Mechanism

Balance reduces the collateralisation ratio in steps, by using layers. The layer mechanism follows the Layered Behaviour-Curated Registry (LBCR) design pattern [49]: an agent can only be in one layer at a time, higher layers mean higher rank, and each layer has its own membership rules. Agents advance to higher layers by performing desired actions, which have scores associated to them. Each layer has upper and lower threshold scores, which decide if agents are demoted, advanced, or left in the same layer.

Drawbacks

From a privacy perspective, a shortfall of Balance is that it encourages agents to have long-lived identities and thus anyone can observe their financial activity in DeFi.
Another drawback is that good and bad behaviours need to be explicitly defined and scored, which makes adoption non-trivial.
Furthermore, it is not an ideal solution for reducing collateral when minting stablecoins, where speculators act as market makers and ensure the coins are pegged to their target value [50]. In such a scenario, desired and undesired actions change very frequently and their scores would need to be dynamically adjusted.

2.5.2 Compound

A non-custodial protocol for lending and borrowing, Compound uses liquidity pools as opposed to peer-to-peer interactions [51]. The interest rate is adjusted based on supply and demand forces.

2.5.3 Aave

Similarly to Compound, Aave allows users to lend and borrow in a non-custodial manner, but its distinctive feature is flash loans. Agents can take under-collateralised loans as high as the liquidity in an Aave pool, with the requirement of paying it back in the same transaction, plus interest rate (currently 0.09%) [52].

2.5.4 Maker

This is the biggest DeFi protocol in DeFi by collateral. Its main ”products” are Dai, a stablecoin, and the Maker (MKR) token, used for governance [53]. MKR tokens are used for voting, owning one token being equivalent to having one vote. The Dai peg to the USD is maintained using a Stability Fee and the Dai Savings Rate. The former is identical to the interest rate of a loan, suggesting that every Dai is borrowed, not minted forever. The latter is used to stimulate demand in the token.

2.5.5 Uniswap

Decentralized exchanges enable users to trade assets without trusting the platform as a custodian. Uniswap achieves this by having liquidity pools for asset pairs, which are used to intermediate the trade of any two assets. Agents can become liquidity providers by depositing both assets that a pool consists of. In exchange, they receive rewards paid from the exchange fees [54].

2.5.6 Synthetix

Trading fiat currencies, commodities and more intricate financial products are made possible in DeFi through synthetic assets [55]. To create a synth, just as for Dai, one has to stake SNX, the Synthetix Network Token. Since SNX is less liquid than ETH, stakers need more collateral to account for the risks discussed in Section 2.7.1. Synthetix also offers a decentralized exchange that is different from the peer-to-peer system of Uniswap, rather being peer-to-contract. The debt of one synth is automatically transferred to other synths if need be, thus providing infinite intra-synth liquidity.

2.6 Trust and Reputation Systems

In the online environment, agents often interact without knowledge of each other’s identity or level of integrity. Without a system to promote honest cooperation, there is little reason for one peer to trust another and thus interact efficiently. Both peers would be exposed to a high risk of the other not ”playing by the rules”.

When it comes to service providers and consumers, the amount of risk they are exposed to is asymmetrical. Consumers have to accept the risk of paying for a service or good before receiving them, which may end up being sub-standard [56]. Service providers, on the other hand, receive the payment instantly. The only way to encourage the actors in this scenario to act honestly is by rewarding positive actions and punishing negative actions.

Trust and reputation have proven effective in online platforms, such as stars and reviews on Amazon, where users explicitly rate their experiences [57]. Even if a customer cannot try a product beforehand, they will be able to better discern the risks of buying it. Moreover, sellers gain a competitive edge by receiving positive reviews and lose sales in the other case.

2.6.1 Trust

Notwithstanding its effectiveness, trust is difficult to measure given the limited amount of information a platform has about its users [56]. In real life, there are a plethora of cues that can guide someone's decision of trust.

Formally, McKnight & Chervany (1996) [58] define trust as *"the extent to which one party is willing to depend on something or somebody in a given situation with a feeling of relative security, even though negative consequences are possible"*. This definition outlines three facets of trust, which are dependence, utility and situational risk [56]. The first party *depends* on the trusted party for a service, the delivery of which may cause positive or negative *utility*, all while the first party deals with a certain probability, or *risk*, that the transaction will not go as intended.

2.6.2 Reputation

Reputation is closely related to trust, the difference between the two being that the former is an aggregated or collective measure of the latter, which occurs on a subjective basis. Thus, on an individual level, one may say "I trust you despite your bad reputation" [56], a statement which implies that the decision of trust is made with some additional, private information. This shows that the opinions of others are usually considered only in the absence of direct experience. In consequence, trust can be established by explicitly applying transitivity, whereas reputation uses transitivity implicitly, through aggregation.

Resnick et al. (2000) [59] suggest that there are three fundamental properties a reputation system must meet in order to be effective:

1. Past interactions must be used to inform decisions of trust
2. A requirement of Rule 1) is that agents must be encouraged to be long-lived
3. Another requirement of Rule 1) is that agents must be willing to give and receive feedback

There are challenges that the properties above do not solve. One is obtaining honest feedback, since agents may be coerced or into providing dishonest feedback or to not give it at all. Second, game theory has been used to show that "cheap" pseudonyms, as is the case with Ethereum accounts, reduce the effectiveness of reputation systems [60]. Rule 2) addresses this partially, but agents might still use Sybil identities to boost their reputation [57]. Third, research in economics has shown that a reputation system can only impose limits on how much deception can be avoided, given that there is a conflict between the cost of building a good reputation and the benefits it brings [61], which promote fluctuations in reputation; however, such erratic behaviour can be minimised when recent actions are given more importance [62]. Fourth, discrimination can occur between agents, which may be difficult to identify [56]; in this case, the feedback of the discriminating agent should not be allowed to impact the victim. Fifth, the reputation system should take accuracy (or lack thereof) into account (e.g. a single positive review is less informative than five positive ones and one negative review). Sixth, it should be expensive to perform ballot box stuffing, or acquiring feedback through cheap, repeated actions [56].

Reputation Context and Web of Trust

Most often, reputation is context-dependent [63]. One should not assume that a good scientist is a good sportsman. But assuming a good scientist is a good teacher seems more plausible. This example shows that in spite of context-dependency, there appears to be a degree of overlap between contexts. A more granular reputation system, one that scores agents in multiple contexts, can be more accurate, but it is more computationally expensive and error-prone. The high likelihood of error occurs because assigning a numeric value to context overlap is an imprecise science. Web of trust systems aim to combine trust transitivity and measures of context overlap in order to compute a global reputation score. In the literature, such systems have also been called ”Transitive trust with semantic constraints” [64].

General Trust Models

Be it ”Web of trust” or some other name, such systems aim to compute a general form of reputation. For instance, REGRET (Sabater & Sierra, 2001 [65]; Sabater-Mir, 2003 [66]) combines multiple facets of trust into what they call an *ontological dimension* of reputation. This ontological measurement is used for general concepts, but the way it is computed differs from one individual to another, depending on their preferences. More specifically, the system is modelled as a Direct Acyclic Graph (DAG). Then, the reputation of each node is the aggregation of its children’s reputation scores. The structure of the DAG and weight of each parent-child edge may differ based on subjective preference. An example the authors give is the reputation of an airline, which can be observed in Figure 2.4. Of course, someone else judging the reputation of the airline may not even consider on-board food quality as a criterion.

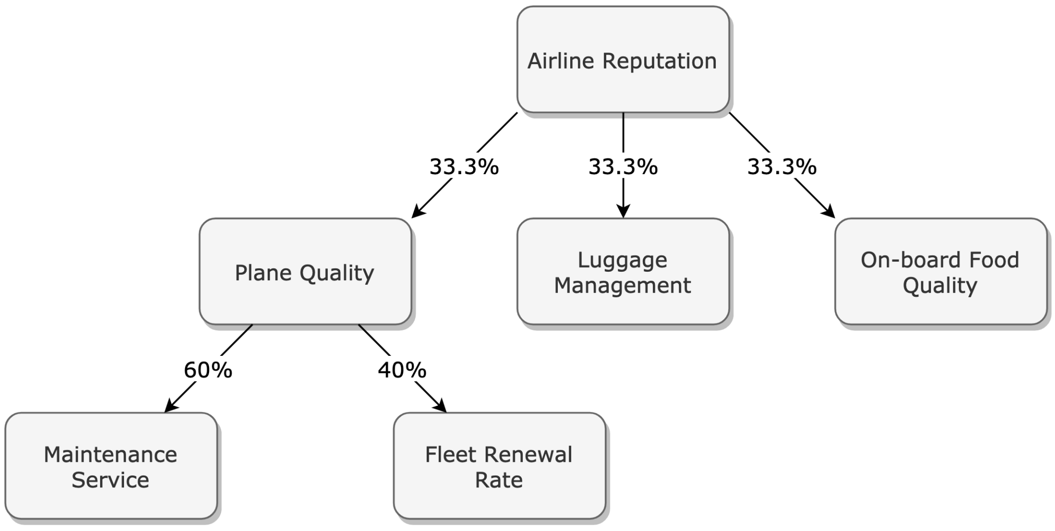


Figure 2.4: Example of ontological reputation of an airline. As the measure is subjective, the components comprising the reputation and their respective weightings can differ according on the evaluator.

Reputation in Cryptocurrencies

Classical PoW consensus algorithms have been criticized for the fact that their security is only based on empirical evidence, since at any moment someone could rent 51% of the mining power in the network and rewrite history [67]. A recent paper exemplifies using mining history, instead of instantaneous hash rate, as a miner's power. In other words, mining power is the sum of all the PoW work done by a miner over time, also named Proof of Reputation. Considering that the network hash rate is constant, after just one year of activity, such a currency would be resilient against a 51% attack that lasts less than a year. The paper goes on to show that there is no incentive for an economically rational agent to attack the network. RepuCoin, the cryptocurrency introduced by this work, also has better throughput than Bitcoin: 10000 transactions per second as opposed to just 7 [67].

2.6.3 Decentralized Trust Management

Web of trust, with its general trust considerations, is an instance of Decentralized Trust Management (DTM) [68]. Such a framework acts as a standard for configuring security policies, authorising credentials, and managing relationships between actors in security-critical operations. Before DTM was established, distributed trust was in use, but was implemented in specialised ways for specific purposes, such as for X.509 and PGP. DTM was introduced as a general tool for tackling all trust management needs, just as SSL was introduced as a standardised way for securing network communication [69].

A flaw in early DTM implementations such as PolicyMaker [68] is that the trust relationships it defines is static [70]. It can only be configured by the administrator of the system and cannot adapt to the changing attitudes of the agents in the system. Feedback and new experiences are not aspects PolicyMaker uses to model trust relationships. While such a framework serves email security, for example, excellently, it is too rigid for more dynamic types of decentralised applications, as is the case in DeFi. Given DeFi is still in its early days, Dapps and their specifications are in constant change. If PolicyMaker were to be used for a web of trust between Dapps, it will probably become outdated in no time. Thus, there is a need for a DTM framework which allows for trust re-evaluation (from feedback) and for dynamically redefining actor roles and relationships.

2.7 Security

2.7.1 DeFi Vulnerabilities

Setting out on a journey to replace traditional finance is no easy task, even when starting with a clean slate and cutting edge technology. It has been shown that in its current form, DeFi is vulnerable to a number of threats [5].

Hijacking Maker Governance

Maker is the number one Dapp by locked-in collateral. It holds 55% of the $952M in DeFi [71]. The failure to protect its user's funds would not only result in massive financial losses, but would also severely undermine the trust in DeFi. Most famously, Maker issues DAI, a stablecoin algorithmically pegged to the USD.

Because decisions related to the governing of the system need to be taken, Maker also issues an MKR coin, the holders of which have voting power proportional to the amount of MKR they hold. MKR is used for *executive voting*, the result of which is a newly elected executive contract which defines the rules Maker enforces.

The design of this governing mechanism has an inherent risk of undergoing a hostile takeover, where the attacker obtains enough MKR coins to have the majority vote and gains control over all the collateral in Maker. The most notable prevention mechanism against such an attack is the *Emergency Shutdown*, which suspends the system given sufficient MKR is used when voting. This protection is effective against a slow hostile takeover, such as crowdfunding led by rebellious DeFi-ers. However, given the emergence of flash loans, a hostile takeover can happen in a single transaction given a malicious executive contract is already deployed. Such a flash attack needs enough liquidity, but it is risk-free from the attacker's perspective, since if anything goes wrong the transaction can be reverted.

A potential defence against this attack would be using a voting mechanism like RepuCoin's Proof-of-Reputation [67]. If a holder's vote weighs as much as the sum of amounts they held over time, flash attacks are rendered useless even if the attacker flash-buys 100% of the MKR tokens.

Price Drops

Price drops, like the one in Figure 1.1 can cause a DeFi lending protocol to become under-collateralised and thus have more debt issued than underlying collateral. In this scenario, given agents are not legally bound to act with integrity, the economically rational decision is to default on their loan.

Besides asset volatility, liquidity also impacts the security of a lending protocol. Because liquidity means how much of an asset can be sold on a market without changing its price [72], selling collateral in an illiquid market will dramatically impact its price negatively. Gudgeon et al. (2020) [5] show that regardless of collateralisation ratio, illiquidity may result in the lending protocol itself defaulting. For instance, with ETH collateral worth $750M USD and a somewhat illiquid market, it would take 40 days for the debt to become higher than the collateral.

Financial Contagion

Since assets generated by one DeFi protocol can be used as collateral in other protocols, the defaulting of one Dapp will severely impact the stability of other Dapps. Moreover, an agent holding an under-collateralised asset will naturally try to buy assets which are uncorrelated in order to reduce risk. This spreads the failure to all assets that are traded in exchange for the low-quality asset, and potentially outside of DeFi as well [5].

2.7.2 Verification Methods

Simulation

Section 2.6.2 discussed some of the basic attacks on trust and reputation systems (TRSs). Yet even when there are mitigations in place against such attacks, the increase in Artificial Intelligence (AI) capabilities brings new challenges to system security [73]. New attacks may no longer need human labour and become cheaper as a result. The sophisticated nature of AI may reveal highly effective and targeted attacks, which are impractical for humans to execute and unlikely to discover using the current tools. Most likely, the attacks that are going to flourish in the AI era of computer security are *strategic attacks*. While single-actor attacks have been explored comprehensively in the literature [74, 75], it is when multiple actors coordinate their actions in order to achieve malicious goals that they create a strategic attack [76]. Gunes et al. (2019) [76] proposed a framework for identifying coordinated attacks, which models attack goals as optimisation problems and looks for maxima in this space through Monte Carlo Simulation and Hierarchical Sampling. The framework has found vulnerabilities in all analysed systems and could serve as a rigorous evaluation method for Byzantic.

Game Theory

Rather than simulating various actions that can be undertaken by an agent, game theory attempts to prove security assumptions using sound deductive arguments [77]. There foundation concepts for any game-theoretic proof are: *game*, *economic agents*, *utility*, *economically rational*, *simultaneous-move*, *sequential-move*.

The system under analysis is modelled as a *game*, where *economic agents* take decisions which influence the final result. By definition, an agent has a set of preferences that they intend to fulfil, and doing so increases their *utility*, which is a measure of subjective welfare. Agents employ strategies to reach their goals. If an agent is able to rank outcomes by preference, understand the steps needed to reach an outcome, and pick, at every step in the game, the most suitable action in terms of reaching the best outcome, it is said to be *economically rational*.

A game may either be a *simultaneous-move* one, or a *sequential-move* one. An example of the former are businesses, where strategic moves may occur at the same time, while an example of the latter is chess. When an outcome is reached based on the player's decisions, the game is said to have reached an equilibrium.

A fundamental flaw in game theory, as in economics, is the assumption that individuals are self-interested and utility-maximising. In cases where individuals are guided by ideology or feelings, they may not be seeking to maximise economic utility.

2.7.3 Attacks Relevant to Byzantic

Network-Level Attacks

Double-spending attacks have been broadly explained in section 2.1.2. These take advantage of the lack of finality in blockchain to send a transaction and then renege it using another transaction that spends the same coins. The attacker gains utility as if both transactions are final.

In *selfish mining*, miners hide (do not publish) new blocks in an attempt to earn more [78]. If they are able to keep a private fork that is longer than the main public chain, publishing this fork would invalidate the PoW of the network from the block at which the fork occurred onward. Rational miners will join the pool controlled by the selfish miners, potentially compromising blockchain decentralisation.

Transaction reordering happens when transactions that were launched in a certain order are added to the blockchain in a different order (i.e. as part of different blocks). This may happen either because miners prioritise certain transactions unfairly or because some transactions offer a higher reward than others. This attack can be used to front-run transactions and create an arbitrage situation, forcing the sender of honest transactions to pay more than normal. An example is in a Decentralized Exchange, where traders placing a buy transaction cause front-runners to buy all the liquidity first, by placing transactions worth much more gas than regular transactions [79]. Front-runners then do sell the assets to the traders, but at a higher price.

Application-Level Attacks

Protocol agents can attempt to *single-shot* their reputation: after gaining a certain reputation, they can take full advantage of it through one malicious action. However, proper game-theoretical incentives should make agents gain less utility from exploiting their reputation after gaining it first [11]. If agents do not wish to single-shot, but instead stay in the protocol and exploit their reputation, their behaviour can also be discouraged with a proper incentive structure [62], as discussed in 2.6.2.

By using *sybil identities*, agents can interact with themselves and falsely boost their performance [57].

Since in Ethereum an account is not tied to formal identity, the reputation agents gain can be sold [57]. However, since private keys cannot be changed, the seller will still be in control of the account.

Compositional trust: agents may try to use the same collateral for multiple loans. If they become under-collateralised using this technique, the best strategy for them is to default.

Code bugs. The most common one is reentrancy - if a function controlling funds can be called again before it ends, the contract can be drained of funds by someone who exploits this property. Others are integer overflow and underflow and being dependant on block timestamps (which can be changed by miners) [80].

A considerable challenge of blockchain is accessing off-chain data securely. The providers of such data are called Oracles, and while the blockchain is secure, any information coming from the outside can potentially be altered [81].

2.8 Technology

2.8.1 Solidity

Programmers have two possibilities for writing and deploying smart contracts on the Ethereum blockchain. They could do so by using EVM bytecode directly, but that would be rather unproductive. A simpler alternative is using a high-level language such as Solidity, which is object-oriented and borrows from popular languages such as C++ and JavaScript [82].

2.8.2 Challenges

As Ethereum was launched only five years ago, it still has a long way to go before it reaches maturity. Zou et al. (2019) [83] outline five challenges to smart contract development that this project needs to overcome.

Security

Because they are dealing with people's funds, smart contracts have very high security requirements. An obstacle to fixing bugs once found is that transactions are irreversible and implicitly code is unmodifiable after development. As a workaround, the proxy design pattern is widely used in order to handle contract versioning [35]. The fact that deployed code is public means that attackers can easily discover any vulnerabilities and carefully craft their exploits.
Solidity compilers are very young and buggy. A good comparison are C language compilers, where hundreds of bugs have been found even 39 years after the initial publication of C [84].

Debugging

Debugging is utterly painful in smart contracts, especially when they call external, on-chain contracts. The main reason is the lack of interactive debuggers, the most advanced of which is the Truffle Debugger, whose interface is similar to GDB [85], yet cannot even run through code stored on-chain. What is more, most error messages are barely informative (both in Solidity and the EVM). Being able to observe the Solidity stack trace of a crash is still not a possibility today, with the exception of using Buidler on local contracts [86]. Printing logs is also not possible outside the constrained use-case of Buidler.

Solidity

Lack of libraries. Every major programming language incorporates or has access to a wide range of libraries that are well tested. In Solidity, on the other hand programmers need to constantly re-write methods like checking if an element is in an array or string manipulation.
Lack of CI/CD [87]. Another significant impediment to development speed is the lack of automation tools (such as Continuous Integration / Continuous Development) for deploying code to a test environment, even for the local blockchain.
Inability to use configuration files. It has become common sense in software engineering to use different configurations for different tests. It should be possible to use the same contract and different configurations with addresses for testing the Ropsten testnet and the mainnet.

EVM

Critics of the EVM have claimed that the bytecode is too slow and that general-purpose languages cannot be used to develop smart contracts [83]. Moreover, the execution environment is very resource-constrained, and there is a limit on the number of local and global variables as well as on the stack size.

Chapter 3

System Overview

3.1 Description

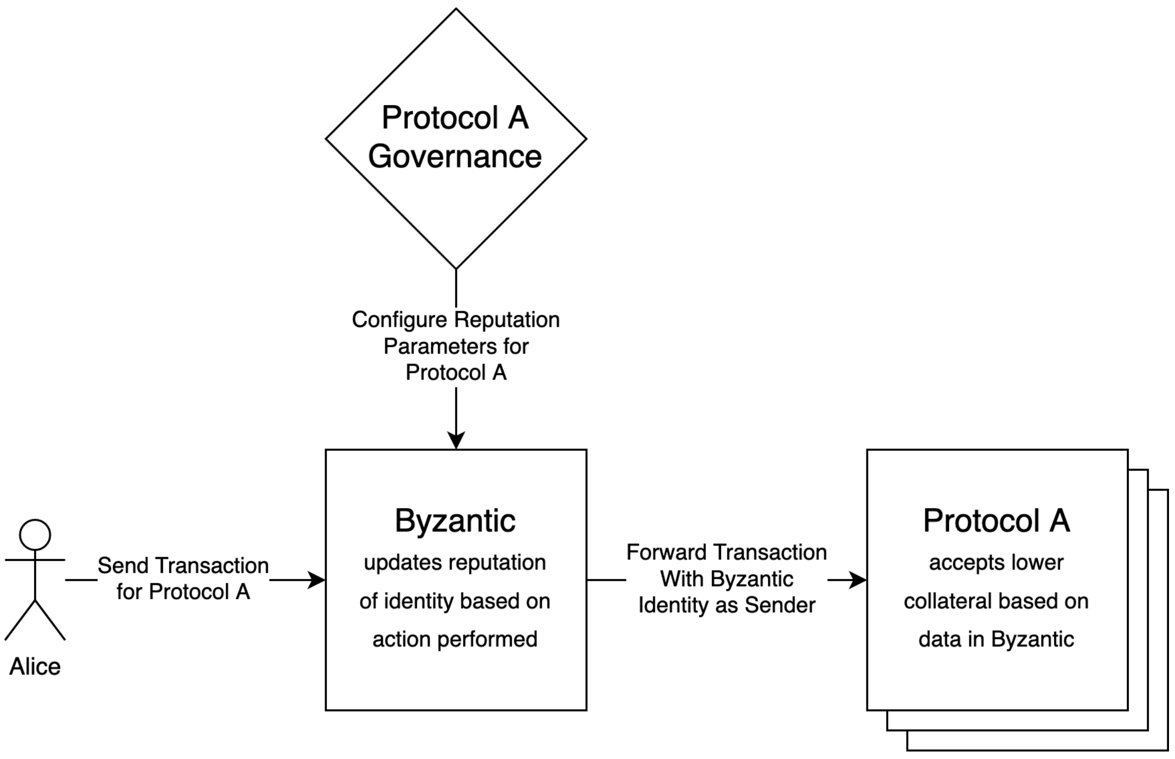

Protocol A Governance
Configure Reputation Parameters for Protocol A
Alice
Send Transaction for Protocol A
Byzantic
updates reputation of identity based on action performed
Forward Transaction With Byzantic Identity as Sender
Protocol A
accepts lower collateral based on data in Byzantic

Figure 3.1: High-level diagram showing how Byzantic intermediates transactions to track reputation. The parameters which define what constitutes a good reputation in Protocol A are configured by protocol governance.

Agents who wish to build reputation encode all state-changing transactions they would have sent directly to DeFi lending protocols, sending them to Byzantic instead. Byzantic keeps a reputation score for every agent, which is updated according to their behaviour. If a user constantly performs ”desired” actions, they receive a reduction of up to 10% in collateral. Desired actions can vary from one protocol to another, and it is protocol governance who configure the scoring of each action.

Moreover, the collateral discount is computed using the user's reputation from all protocols, meaning that new lending protocols can integrate with Byzantic and attract users who already have good reputations elsewhere.

Our system keeps track of user identity by deploying a new contract for every user who registers. As Figure 3.1 illustrates, when Byzantic receives a transaction to forward, it is that "identity" contract which calls Protocol A. If Byzantic were simply to be notified about which actions are taking place, there is a risk of agents only recording high-scoring actions in our protocol, in order to boost their reputation unfairly.

Because reputation needs to be tied to an identity, the design in Figure 3.1 exposes the transaction history of its users. However, if users have access to a trusted Ethereum node that can act as a relay, they can use the DejaVu design pattern described in Section 6 to alleviate this issue. In Figure 3.2, Alice is originating transactions from the internet layer and uses DejaVu to anonymously publish them onchain. Her Byzantic identity can still only be used by her, owing to single-use zero-knowledge proofs.

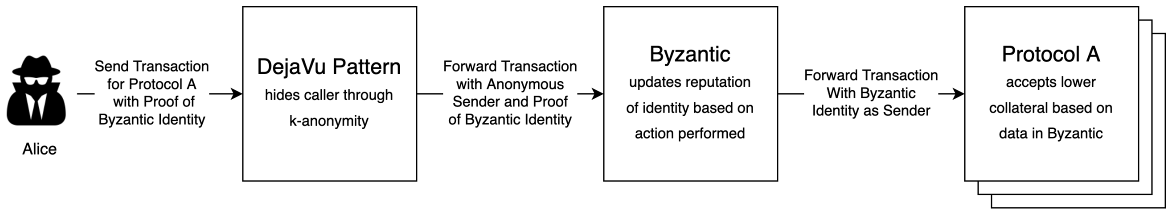

Alice
Send Transaction for Protocol A with Proof of Byzantic Identity
DejaVu Pattern
hides caller through k-anonymity
Forward Transaction with Anonymous Sender and Proof of Byzantic Identity
Byzantic
updates reputation of identity based on action performed
Forward Transaction With Byzantic Identity as Sender
Protocol A
accepts lower collateral based on data in Byzantic

Figure 3.2: High-level diagram of Byzantic integration with the privacy-preserving design pattern DejaVu.

3.2 Actors

We have identified three actor types, with the following characteristics with respect to Byzantic.

- **Agent**. Intends to use good reputation to provide smaller security deposits when borrowing in DeFi. Anonymity is a top priority. Desires to pay low transaction fees. Uses reputation in DeFi for informal purposes to improve economic utility, such as by credit delegation (Section 12.1).
- **DeFi Protocol Governance**. Aims to maintain smart contracts to the highest level of security, and default risk to the lowest possible level. Wants to contribute to liquidity creation in DeFi.
- **Byzantic Registry**. The registry is comprised of two smart contracts deployed on a decentralized ledger. It is entrusted by protocol governance to correctly measure user reputation by tracking their behaviour.

3.3 Requirements

To be considered successful, Byzantic implementations must meet eight criteria.

- **Collateral reduction**, as a means of rewarding well-behaving agents and producing liquidity.
- **Transferable reputation**, so that users who mainly act in a single protocol can receive security deposit discounts in all protocols.
- **Misconfiguration resistance**, to protect DeFi protocols in case one of them updates to an insecure Byzantic configuration, which awards reputation to easily.
- **Price crash resistance**. Our solution must be resilient against ”black swan” events that would cause deposits to quickly depreciate.
- **Transparency**. As Byzantic is a reputation model for public decentralized ledgers, anyone should be able to inspect how it operates.
- **Sybil identity resistance**. Agents should gain no additional utility from using several identities in Byzantic.
- **Strategy proofness**. Byzantic implementations should be bug-free and correctly follow the model specification, so as not to allow users to ”game” the system and boost their reputation.

3.4 Assumptions

We make the following assumptions. First, reputation is transferable only if there is **quantifiable inter-protocol compatibility**. Second, Byzantic needs **distributed ledger functionality** that enables DeFi to exist. Third, debt is issued using **over-collateralisation**, so it can be reduced by measuring reputation. Fourth, agents in DeFi are **economically rational**. Fifth, Byzantic reputation should have a **verifiable specification**, which means that anyone can check the correctness of its results.

Chapter 4

Byzantic

4.1 Main Components

Byzantic uses a *registry* to compute reputation in a twofold way. On the one hand, it uses a Layered Behaviour-Curated Registry (LBCR) smart contract to track reputation at protocol-level. It assigns users into different layers of ”notoriety” in a round-based manner, based on their behaviour in the previous round. The LBCR is described in Section 4.1.1. On the other hand, the registry aggregates information from all LBCRs when determining a user’s collateral reduction. Aggregation is performed at system-level, in the Web of Trust smart contract, detailed in Section 4.1.2. For both of these components, configuration is decided by protocol governance when they integrate with Byzantic.

4.1.1 Layered Behaviour-Curated Registry

Byzantic extends the LBCR introduced in Balance (Section 2.5.1) and is a round-based mechanism for tracking reputation. Every round, users are scored based on the actions they perform. At the end of a round, users are assigned new layer positions based on their ”performance”. Every protocol that integrates with Byzantic has its own LBCR, with the following parameters.

- **Layers**. The number of layers to be used for promoting users.
- **Action Rewards**. The reward users receive, within a round, for performing a certain action. Action scores can be fixed or computed using a function, to account for different transaction values.
- **Layer score boundaries**, which specify the minimum score a user needs qualify for the membership of a layer.
- **Layer Factors**. An array of values between 0 and 1, each value corresponding to a collateral reduction. For instance, a factor of `0.9` on a base collateralisation rate of 150% results in a collateralisation of 135%.

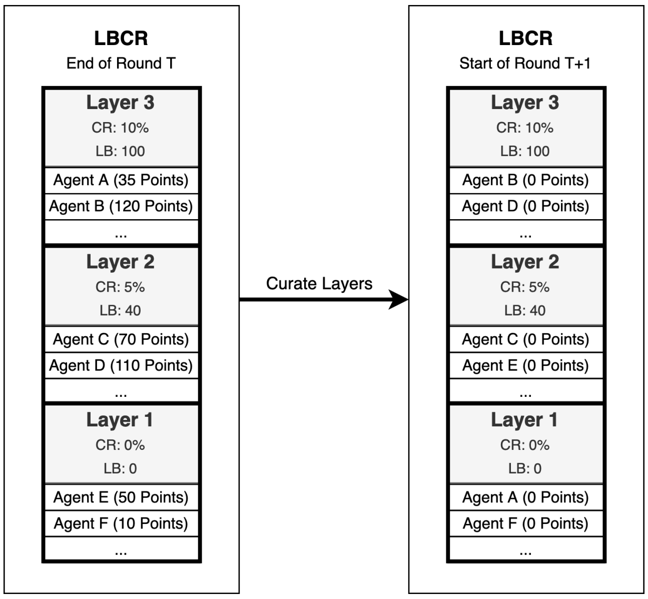


Figure 4.1: Layered Behaviour-Curated Registry (LBCR) smart contract functionality. The diagram on the left-hand side shows the state of the LBCR at the end of a round, with the score each agent has gained during that round between parentheses. Every layer has a specific Lower Boundary (LB) required to be promoted to it; the higher the layer, the higher the reputation a user has and the more Collateral Reduction (CR) they receive. The arrow marks the transition to the next round, during which users are promoted or demoted based on the score they achieved in round T. New rounds reset user scores to zero, so that users need to consistently act in the interest of the protocol to receive high CR.

- **Compatibility Scores**. A value between 0 and 100 that is assigned to every other protocol in Byzantic. It quantifies the transferability of reputation between each of those protocols and the current protocol.

- **maintainCompatibilityScoreOnUpdate**. As a result of using LBCR configuration versioning (Section 5.4), protocol governance should set this Boolean parameter. It represents whether compatibility score with an updated protocol should remain the same after LBCR version updates and its default value is `True`.

4.1.2 Web of Trust

When a DeFi protocol checks whether an agent is properly collateralised, the Web of Trust component is the one being called. It determines agent collateral reduction using one of two functions: *Reputation Maximiser* and *Weighted Average*. Figure 4.2 illustrates the high-level functionality of this component.

Reputation Maximiser

Following the system evaluation in Section 9. This is the aggregation function of choice. Given the inter-protocol compatibility scores and LBCR layer positions of a

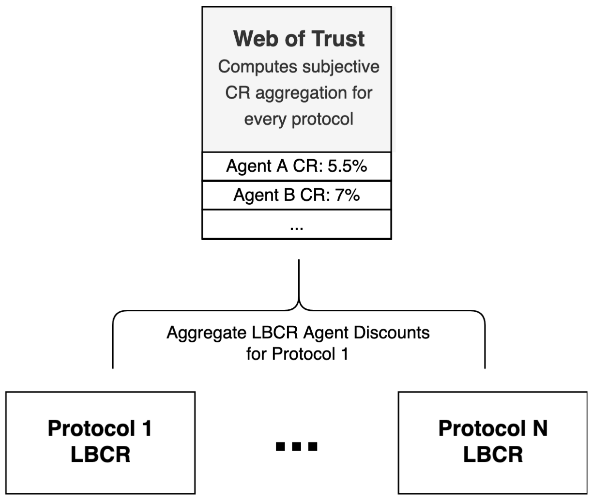

Web of Trust
Computes subjective CR aggregation for every protocol
Agent A CR: 5.5%
Agent B CR: 7%
...
Aggregate LBCR Agent Discounts for Protocol 1
Protocol 1 LBCR
Protocol N LBCR

Figure 4.2: Web of Trust smart contract functionality. To decide the discount that should be awarded to its users, Protocol 1 calls the Web of Trust smart contract to aggregate user reputation across all protocols in Byzantic. Protocol 1 specifies the parameters of the aggregation, so the final result is both comprehensive (combines all LBCRs) and specific to Protocol 1 requirements.

user, this function computes the user's maximum discount. The following python code describes its implementation. Its input, `LBCR_Discounts`, is an array containing `(compatibilityScore, discount)`, where `discount` is the reputation achieved by the current user in a DeFi protocol with `compatibilityScore`.

```
def raputationMaximiser(LBCR_Discounts):
    LBCR_Discounts.sort(key = lambda x: x[1], reverse=True)
    aggregatedDiscountInCurrentRound = 0
    compatibilityScoreRemaining = 1
    for (compatibilityScore, discount) in LBCR_Discounts:
        if compatibilityScoreRemaining == 0:
            break
        compatibilityScoreUsed =
            min(compatibilityScoreRemaining, compatibilityScore)
        compatibilityScoreRemaining -= compatibilityScoreUsed
        aggregatedDiscountInCurrentRound +=
            (compatibilityScoreUsed * discount)
```

Weighted Average

Let R_{A,P_k} denote the reputation Byzantic will compute for agent A with respect to protocol P_k, and C_{P_x,P_y} to denote the inter-protocol compatibility of protocols P_x and P_y. Given n protocols integrated with Byzantic, the *Weighted Average* function uses the following formula to determine R_{A,P_k}:

$$R_{A,P_k} = \frac{\sum_{i=0}^{n-1} R_{A,P_i} \times C_{P_k,P_i}}{\sum_{i=0}^{n-1} C_{P_k,P_i}}$$

The literature proposes two improvements to the approach above.

1. **Time-Weighted Average**. This approach is supported by game-theoretic analysis [62]. However, it was not implemented in Byzantic, because Section 9.3.1 showed it is detrimental to the system responding promptly to price crashes. Instead of using the current layer position of an agent A in protocol P, we can take into account past layer positions when computing $R_{A,P}$. For instance, if we use a forgetting factor F, and we take into account the last k layer positions, out of the total N layer positions:

$$R_{A,P} = \sum_{t=0}^{k-1} R_{A,P,N-t} \times F^t$$

   Informally, this can signal whether an agent ever lost their position from the top layer of trust, even if they are currently on the top layer.

2. Ontological reputation using a Direct Acyclic Graph, as shown in Section 2.6.2. However, the reputation graph of Byzantic forms an undirected graph, so it is incompatible with this proposal.

4.2 Initial Implementation Approach

The approach to implementing Byzantic evolved over time, as certain design limitations became apparent. The initial implementation aimed to accurately track reputation and, in theory, to allow users to lock in less collateral. To avoid making code changes in target protocols, Byzantic required low reputation agents to provide even more collateral than would normally be required. It would then redistribute the difference between the standard collateral requirement and the increased Byzantic requirement to high reputation agents, such that their security deposit would be reduced. Nonetheless, low reputation agents can always withdraw the full amount deposited at any time. This may prompt Byzantic to repay the loans of some high-reputation users to retrieve the redistributed funds of the low-reputation user. This approach had the great advantage of not requiring any code changes in target protocols, but it did not produce any liquidity.

4.3 Second Implementation Approach

The initial implementation showed that collateral redistribution cannot lead to liquidity creation. It made it clear that liquidity can only be added to DeFi if protocols trust Byzantic and allow high-reputation users to provide less collateral. However, it still seemed possible to achieve this without modifying already deployed smart contracts.

4.4 Final Implementation Approach

Considering that the first implementation (Chapter 5.2) could not generate liquidity and the second one was technically infeasible, we evaluated the trade-offs inherent in implementing Byzantic. Similarly to the pivoting concept from Lean Startup principles [88], we needed to steer the project in a new direction. We identified two alternative pathways: offering interest-free loans and unlocking liquidity. The former does not require code changes, while the latter does. Ultimately, we chose the latter, as lack of liquidity is a more pressing problem in DeFi at the moment [5].

Interest-free Loans. Without modifying the code of deployed protocols, Byzantic would be best suited as an extension to the initial implementation - performing re-distribution of funds, but for general purposes. High reputation agents could use Byzantic buffer funds not only for collateral, but also for flash loans or other use cases. Effectively, Byzantic would offer interest-free loans to high-reputation users. These loans would only be usable through a Byzantic address, for operations that can be ”reverted” within a transaction by Byzantic (deposits, flash loans, possibly even trading on a decentralized exchange).

In the unlikely case where all agents are in the same LBCR layers, they would all have a Byzantic factor of `1.0`, so there would effectively be no low-reputation agents, since all agents have the same reputation. Thus no excess funds would be available to loan.

Interest-free loans are a novel concept in DeFi. A drawback is that the duration of a loan is unknown in advance, as it depends on there being buffer collateral in Byzantic. In the worst-case scenario, the minimum duration of such a loan is one transaction, which is comparable to flash loans. Unfortunately, this solution does not add liquidity to DeFi.

Unlocking Liquidity. To add liquidity to DeFi, collateral redistribution can no longer be considered. Protocols that integrate with Byzantic need to allow high-reputation agents to use less collateral, by trusting Byzantic. As low liquidity may pose an even greater threat to DeFi security than the small code changes required to use Byzantic [5], the final architecture prioritises liquidity creation over interest-free loans.

4.5 DejaVu Design Pattern for Anonymity

The DejaVu pattern allows users to build reputation while staying anonymous. Besides anonymously tracking reputation, DejaVu can also be used for privacy-preserving joint accounts, such as organisation accounts where employee privacy is preserved and the business of the company is transparent.

This solution is made possible through zero-knowledge proofs (Section 2.3.1), mixers (Section 2.3.2) and blockchain-layer relays (Section 2.3.3). Zero-Knowledge Proofs (ZKP) have been used for authenticating transactions to the User Proxy contract (see Section 4.5.1), which is how agent identities are tracked in Byzantic. ZKPs were implemented using ZoKrates, which provides a simple abstraction over ZK-SNARKs. Any mixing service could be used, but this project considered Tornado Cash (Section 4.5.2) because of its simple integration; Zether [89] or Aztec 2.0 [90] (not yet released) can also be used to make anonymous value transfers.

We describe the use of zero-knowledge proofs in Section 4.5.1 and that of a payment mixer in Section 4.5.2.

4.5.1 Transaction Authentication with Zero-Knowledge Proofs

A key requirement for preserving privacy is linking incoming transactions to a Byzantic reputation account, even if the transaction was not initiated by that account, through authentication. To achieve this, Byzantic makes use of the common pattern of proving the knowledge of a hash pre-image without revealing it [91]. This zero-knowledge proof pattern resembles providing a password to log in to an account. In the classical implementation, the password hashes of registered users are stored in a database and login password input is hashed and compared to the stored hashes. If there is a match, the user is authenticated - this process is indeed a proof of hash pre-image. However, there are two differences between traditional authentication and Byzantic authentication: the inability to have authenticated sessions on Ethereum, and the necessity to change the pre-image (the password) after every successful authentication.

Lack of Authenticated Sessions

Authentication in Byzantic is performed at action level: a valid zero-knowledge proof allows for one action to be performed on behalf of a Byzantic account regardless of the caller. Still, because of how Ethereum was designed, this authentication cannot start a ”user session” that spans across multiple actions. Authenticated sessions over the web are technically feasible because the server sends a secret cookie over an encrypted channel. In a perfectly secure implementation of encrypted communication, only the user has access to the secret cookie and can use it to maintain an authenticated session. In Ethereum, however, all transaction data is public, so if a session cookie were to be generated, anyone could read and use it. Moreover, password-based authentication is usually not required in Ethereum, because `msg.sender` addresses cannot be spoofed; knowing who the sender is is usually enough to authenticate them.

In the Byzantic privacy model, however, users send a transaction at the internet layer to a relay, which then creates the transaction and becomes the tx.origin. If the `msg.sender` identity was used to create a session based on the valid zero-knowledge proof, anyone using the same relay could hijack the authenticated Byzantic reputation. Thus, at the expense of lower throughput and increased gas costs, reputation authentication is performed for every transaction.

Single-use Passwords

All transaction data is public in Ethereum. Thus, if a user sends a transaction with a zero-knowledge proof to take a certain loan, someone else could reuse the same `msg.data` to take the same loan on behalf of the same reputation and increase the risk of liquidation. Reusing zero-knowledge proofs this way is just like double-spending some funds (Section 2.7.3). To prevent this from happening, users must change the hash value whose pre-image is to be proved after every authentication. Reusing the same `msg.data` results in one valid proof and one invalid proof.

Front-running attacks are a subcategory of transaction reordering attacks (Section 2.7.3), where someone sends a transaction with more gas than a pending transaction, such that the former is mined more quickly than the latter. In this case, an adversary may try to duplicate a Byzantic transaction that contains a zero-knowledge proof authenticating some action. These attacks are not an issue, since the password that is newly set by the first transaction will invalidate the second transaction (which will try to authenticate with the now obsolete hash pre-image proof). The timing does not matter much, since the same action that the user intended to do happens, and the front-runner earns nothing, because they still have not learned the hash pre-image and cannot generate a proof of their own.

Zero-Knowledge Proofs using ZoKrates

ZoKrates is a suite of utilities for zero-knowledge proofs on Ethereum. It implements the proofs using ZK-SNARKs (Section 2.3.1), which is ideal because the proofs are non-interactive. ZoKrates provides a high-level language for writing verification programs that are converted into arithmetic circuits that can be run on-chain. Its standard library includes hash functions and elliptic curve cryptography, which are also compiled to efficient arithmetic circuits.

4.5.2 Tornado Cash

Tornado Cash is a non-custodial payment mixer for Ether and ERC-20 tokens, which works as described in Section 2.3.2. It is needed for transferring funds from an account to another without linking those accounts through a transaction.

Stealth addresses are not a viable solution in this case, because the transactions always happen between two account of the same user: the one that is directly linked to a reputation and another one, which must remain unlinkable to the first account. If a payment was made from the ”unlinked” account to a stealth address belonging to the reputation account, the stealth address would still need to deposit the funds in the reputation account - in which case a stealth account is ineffective at preserving privacy. Mixers, on the other hand, attempt break traceability rather than trying to avoid the problem. Using a mixer like Tornado Cash, payments to the reputation-linked account will always be anonymous if executed diligently enough. See Section 6.3 for issues that users should be aware of when using a transaction mixer.

Chapter 5

Byzantic Implementation

5.1 Technology Choices

5.1.1 Truffle

At the moment, Truffle is the only tool available for stepping through smart contract code. Not only does it allow debugging local contracts, but with some work it can be used for debugging external contracts too (Section 5.5). The debugger is very slow and it can only run tests written in plain JavaScript, but for the most part there are no alternatives.

5.1.2 Buidler

The one case where Truffle can be replaced is when building a project locally. In this case, the much better choice is the Buidler EVM, which is a local Ethereum instance - a completely new blockchain, whose single block is a new genesis block. It has enhanced debugging capabilities such as `console.log`, detailed stack traces and is significantly faster than Truffle.

5.1.3 TypeScript

Most DeFi tools are written in TypeScript and are only compatible with JavaScript [92]. TypeScript is safer to use than regular JavaScript and most IDEs offer autocomplete when using TypeScript. Furthermore, code is easier to understand and extend when written in this programming language. Becoming familiarised with it was useful when looking through the code of those tools (Section 5.5). While Buidler is able to run smart contract tests written in TypeScript, to run the same tests with Truffle one must first compile them to JavaScript, which slows down the development process. To work around this, when we needed to debug transactions to external contracts we also wrote small JavaScript tests and ran them from the Truffle Console.

5.2 Initial Implementation

Byzantic deploys a ”personalised” contract for every (agent, DeFi protocol) pair. Agents deposit funds in such a contract and use those funds as collateral or for other purposes when interacting with a DeFi lending protocol. Users need these individual contracts (named `ProtocolProxy`) because they offer unique addresses, or identities, for interacting with DeFi protocols. Without a unique address, Byzantic would not be able to intermediate transactions while keeping secure against user activity outside of this system.

5.2.1 Architecture

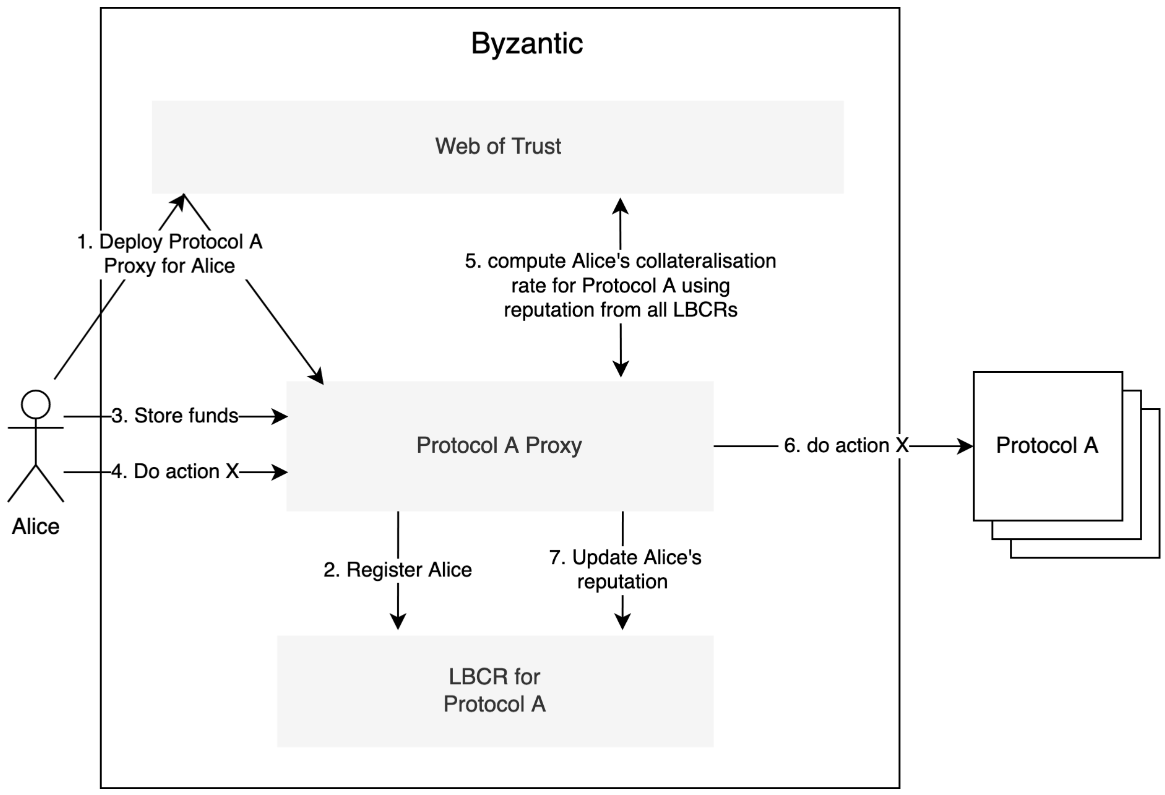

Byzantic
Web of Trust
1. Deploy Protocol A Proxy for Alice
5. compute Alice's collateralisation rate for Protocol A using reputation from all LBCRs
3. Store funds
4. Do action X
Protocol A Proxy
6. do action X
Protocol A
Alice
2. Register Alice
7. Update Alice's reputation
LBCR for Protocol A

Figure 5.1: Initial system architecture. While this solution reduces collateral for high-reputation users, it does not add liquidity to DeFi, which is the main goal of Byzantic. This is because it does not require making changes to the smart contracts of Protocol A.

The **Diagram Components** in Figure 5.1 are the following.

Web of Trust. The core contract of Byzantic. It computes agent reputation using LBCR data from all protocols and deploys Proxy contracts for new users. It is also used to move collateral from the over-collateralised to the under-collateralised.

LBCR. Layered Behaviour-Curated Registry (LBCR) is the mechanism used to promote or demote agents in layers. The higher the position in LBCR, the smaller the collateralisation ratio becomes and agents lose less opportunity cost. The LBCR contract being used is the one from Balance [11].

Protocol A Proxy. A contract that is deployed individually for each user (using the factory pattern) from the Web of Trust contract. The contract mediates and tracks agent interactions with the target protocol, feeding them to the LBCR contract to update the agent score. The Proxy contract can also be used by the user to request more collateral if they are in a high LBCR layer. The additional collateral is taken from over-collateralised agents in the low layers. Whenever a low-reputation user wants to withdraw funds, the Proxy requests any missing funds from the Web of Trust contract, which liquidates some of the buffer collateral from the high reputation users.

Protocol A. DeFi protocol that Byzantic has integrated with (i.e. a Protocol A Proxy has been deployed in Byzantic, and LBCR parameters for Protocol A have been determined).

Alice. The agent that interacts with Protocol A.

5.2.2 Interaction Flow

When a new agent (Alice) starts using Byzantic:

1. Alice calls the Web of Trust contract to deploy a Protocol A Proxy instance for herself. The Web of Trust does so, using the Factory design pattern.

2. The Protocol A Proxy registers Alice to the LBCR for Protocol A.

3. Alice stores funds in her Protocol A Proxy contract. These funds can be used as collateral for loans, or any other purposes related to Protocol A.

4. To interact with Protocol A, Alice calls the Protocol Proxy to execute action X on her behalf.

5. The Protocol Proxy queries the Web of Trust contract, which aggregates the reputation of Alice from all protocols, to decide the amount of collateral needed for action X. If Alice does not have enough collateral, the transaction is reverted at this step.

6. The Protocol Proxy makes the call for action X

7. If the call succeeds, Alice's score is updated in the LBCR of Protocol A.

5.2.3 Integration with Aave

As part of the New York blockchain week hackathon [93], we integrated Byzantic with Aave using the initial architecture (Figure 5.1) and a newly implemented Protocol Proxy for Aave.

Desired and Undesired Actions

Specific action rewards were left to be decided by Aave's governance, so the integration used arbitrary scores. The following desired and undesired actions have been identified, assuming that increased liquidity is the top priority.

- **Lending**
  - **Desired**: Minting aTokens (increases pool liquidity).
  - **Undesired**: Redeeming aTokens (reduces pool liquidity).
- **Borrowing**
  - **Desired**: Repaying a loan, including others' (increases pool liquidity).
  - **Undesired**: Borrowing (reduces pool liquidity).
- **Liquidation**
  - **Desired**: Liquidating a user who fell below the over-collateralisation ratio (reduces default risk).
  - **Undesired**: Being liquidated.
- **Flash loan**
  - **Desired**: Taking a flash loan (adds liquidity through the flash loan fee).

Testing and Debugging

The Ganache Command Line Interface [94] was used to run a locally hosted instance of the Ethereum mainnet. Initially, the Infura Ethereum API [95] was used for spawning a Ganache local fork of the mainnet. The unit tests were written in TypeScript. For each type of action in Aave's Protocol Proxy, a unit test would call Aave both directly and through the Proxy. The tests fail when there is a result mismatch of the two calls.
Integrating with Aave meant every transaction called external contracts, and thus Buidler's debugging capabilities could not be used. Instead, we used Truffle Debug with some simplified tests (as Truffle cannot run TypeScript) and ran step-by-step through the smart contract code. Another challenge to debugging was that the free Infura version only supports querying the most recent 128 blocks, which prevents the Truffle Debugger from fetching all the information it needs. Fortunately, the Centre for Cryptocurrency Research and Engineering at Imperial College runs a full Ethereum node at `http://satoshi.doc.ic.ac.uk:8545`. Finally, using this full node

we could successfully run a Ganache instance and debug using Truffle:

```
ganache-cli -f http://satoshi.doc.ic.ac.uk:8545 -a 10 -e 100 -p 2000 -v
-m depart mistake volume quick route family festival wedding pen fork build
gauge
```

The flags in the command above are:

- `-f`: Fork from the given Etherum client at an arbitrary block (default is the latest block).
- `-a`: Number of test Ethereum accounts to generate.
- `-e`: Ether balance in the generated accounts.
- `-p`: Port of the local Ethereum client in localhost.
- `-v`: Output requests and responses to stdout. This flag is very useful for debugging transactions that reverted because it outputs their transaction hash. While successful transactions return transaction details (including the hash) to the JavaScript client, reverted transactions return `undefined`. The transaction hash is useful because it is the only argument Truffle debug takes.
- `-m`: Mnemonic that can be used to authenticate into the generated accounts from MetaMask, a UI browser client.

5.2.4 Challenges

No Added Liquidity. Since Byzantic would only perform a redistribution of collateral from low-reputation to high-reputation users, there would be no liquidity unlocked by integrating target protocols with our solution. On average, the locked-in liquidity would be the same both with and without Byzantic. Some users would provide less collateral just because others provide more. This realisation led to an in-depth analysis of the various ways to prevent user liquidation, described in Section **??**. This drawback means that the only positive impact the current project would make is encouraging users to abide by social welfare norms. But since Byzantic is an opt-in protocol, users would probably have little incentive to use it.

5.3 Second implementation: Exploring chained delegatecalls

5.3.1 Changes from the previous Byzantic version

Previously, agents had to store funds in their personalised Protocol Proxies separately and withdraw from all of them when they left Byzantic. This did not provide a natural User Experience, and now Byzantic offers a single contract for users to deposit to

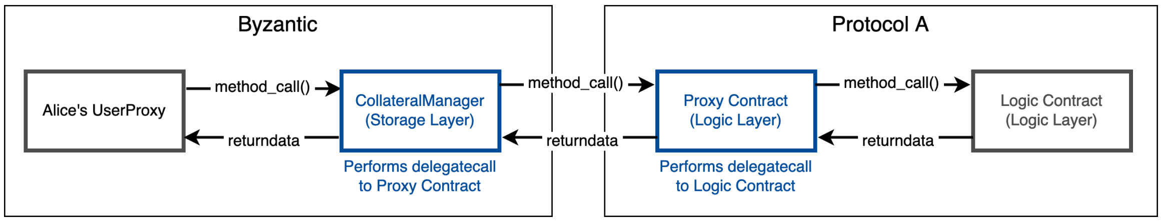

Byzantic
Protocol A
Alice's UserProxy
method_call()
returndata
CollateralManager (Storage Layer)
Performs delegatecall to Proxy Contract
method_call()
returndata
Proxy Contract (Logic Layer)
Performs delegatecall to Logic Contract
method_call()
returndata
Logic Contract (Logic Layer)

Figure 5.2: Chaining of upgradeability proxies that occurs when Byzantic is implemented with another protocol. Because of how `delegatecall` is implemented in the Ethereum Virtual Machine, using it to forward a transaction does not alter the storage of Protocol A. State-changing transactions forwarded this way will revert without a reason.

and withdraw from (the UserProxy). Moreover, a Collateral Manager contract is introduced, meant to become the only way DeFi protocols should allows transactions to happen. This is because it is to be called by both Byzantic and non-Byzantic users, forwarding their calls along with the collateralisation discount they should receive.

A new architecture is introduced, as can be observed in Figure 5.3. The main modification to the event flow is the Collateral Manager contract, an instance of which is deployed for every protocol integrating with Byzantic and intermediates all transactions with it. The Collateral Manager is deployed by DeFi protocol governance, but works as part of Byzantic (see Figure 5.3). This solution requires that protocols completely entrust Byzantic with collateral checks.

Since deposit and borrow details are publicly available in lending protocols via getter functions, the Collateral Manager does not require elevated privileges to compute whether callers are properly collateralised. The Collateral Manager performs user collateral checks, so protocols need to remove their own collateral checks and call the Byzantic Collateral Manager instead.

In order for this solution to work, the Collateral Manager must work as a proxy: if a call is valid, it is ”forwarded” to the corresponding protocol, keeping the `msg.sender` and `msg.value` fields in the call payload unchanged. Therefore, we have attempted to use the Upgradeability Proxy design pattern (Section 2.2.4) to achieve this functionality.

5.3.2 Chaining `delegatecall`s

In Aave and Compound, the main lending protocols that Byzantic is targeting, no logic contracts are directly exposed to users. Instead, they are using the Upgradeability Proxy design pattern (Section 2.2.4). In Aave, when a user requests the address of a contract from the Addresses Provider contract [96], they are returned the address of a proxy to that contract. Thus, Byzantic needs to be designed with the assumption that the aforementioned design pattern is ubiquitous in DeFi.
In the second implementation, Byzantic aims to use the Collateral Manager contract as a proxy to other protocols. However, because all protocols use proxy contracts

”at the edge”, directing all traffic through them, the new architecture unknowingly attempted to chain proxy contracts: Collateral Manager delegatecalls the Proxy Contract of Protocol A, which delegatecalls a logic contract (Figure 5.2). When a transaction that traverses consecutive proxies fails, it reverses without a reason, making debugging very difficult with Truffle. Moreover, using a delegatecall results in state changes in the caller contract, not in the callee contract. In the current implementation of Byzantic, the callee contract is in a foreign protocol. It would be a security vulnerability if it used the storage of the caller contract, making this solution technically infeasible.

5.3.3 Architecture

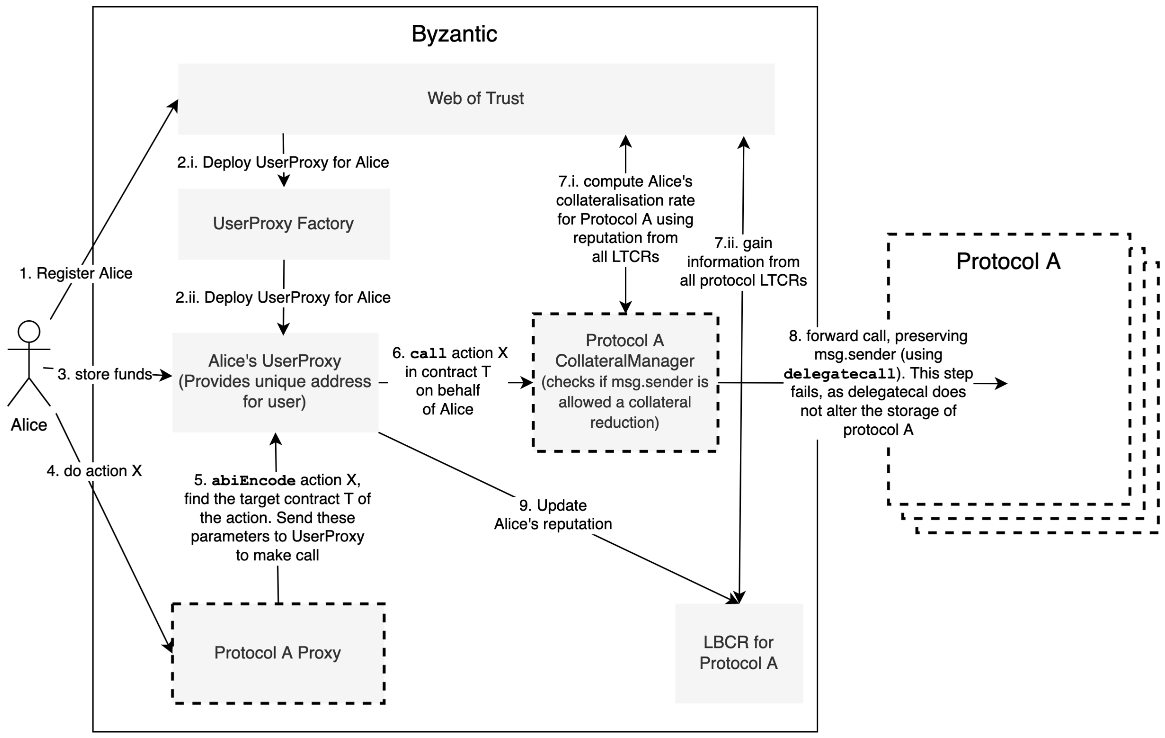

Byzantic
Web of Trust
2.i. Deploy UserProxy for Alice
UserProxy Factory
2.ii. Deploy UserProxy for Alice
1. Register Alice
Alice
3. store funds
Alice's UserProxy (Provides unique address for user)
4. do action X
5. abiEncode action X, find the target contract T of the action. Send these parameters to UserProxy to make call
Protocol A Proxy
6. call action X in contract T on behalf of Alice
Protocol A CollateralManager (checks if msg.sender is allowed a collateral reduction)
7.i. compute Alice's collateralisation rate for Protocol A using reputation from all LTCRs
7.ii. gain information from all protocol LTCRs
8. forward call, preserving msg.sender (using delegatecall). This step fails, as delegatecal does not alter the storage of protocol A
Protocol A
9. Update Alice's reputation
LBCR for Protocol A

Figure 5.3: Second system architecture of Byzantic. Contracts with dashed borders are managed by their corresponding protocol governance.

Diagram Components. We describe how each component in Figure 5.3 changed from the first implementation (Section 5.1) and why.

Web of Trust. No longer deploys unique Protocol A Proxy instances for users, as that contract is now unique per protocol. Instead, `WebOfTrust` calls the UserProxy Factory to deploy a UserProxy contract for each new user. Since Byzantic no longer performs collateral redistribution, `WebOfTrust` no longer has this responsibility.

UserProxy Factory. Deploys a `UserProxy` contract for every new user and stores `(user, userProxyAddress)` pairs.

Alice's UserProxy. This contract is unique to every user and provides a unified address for all user interactions with the protocols Byzantic integrates with. This is in contrast with the initial approach, where Byzantic would call DeFi protocols from different addresses, even if it was the same user originating the transactions. Users deposit to and withdraw funds from Byzantic using this contract.

LBCR. No functionality changes.

Protocol A Proxy. (Not to be confused with the Upgradeability Proxy design pattern). This contract is now unique per protocol and is deployed by the protocol governance, just like the Collateral Manager contract. Due to the introduction of User Proxy contracts, this protocol no longer manages funds.

Protocol A Collateral Manager. New contract that is unique for every protocol and is deployed by the corresponding governance. This contract is meant to be called both by Byzantic and non-Byzantic users, redirecting their calls to DeFi protocol contracts. In order to do this, it implements the Upgradeability Proxy design pattern. Before the call is forwarded, the Collateral Manager checks whether the caller is properly collateralised for the action they are trying to perform.

Protocol A. No longer checks if calls are properly collateralised, as this functionality is outsourced to the `CollateralManager`.

Alice. No changes.

5.3.4 Interaction Flow

When a new agent (Alice) starts using Byzantic:

1. Alice calls the Web of Trust contract to sign up.
2. The Web of Trust contract calls the UserProxy Factory to deploy a personal UserProxy for Alice. She is automatically signed up with all existing protocols in the Byzantic ecosystem. When a new protocol is added to Byzantic, Alice is automatically signed up with it.
3. Alice stores funds in her UserProxy contract. These funds can be used as collateral for loans, or any other purposes related to the protocols integrated with Byzantic.
4. To interact with Protocol A, Alice calls the Protocol Proxy to execute action X on her behalf.
5. The Protocol Proxy bundles the call details in an `abiEncoding` and sends it to Alice's UserProxy.
6. Alice's UserProxy performs a low-level `call` to the Protocol A Collateral Manager, using the `abiEncoding` from the previous step.

7. The `CollateralManager` queries the Web of Trust contract, which aggregates the reputation of Alice from all protocols, to decide the amount of collateral needed for action X. If Alice does not have enough collateral, the transaction is reverted at this step.

8. The call is forwarded to Protocol A using `delegatecall`.

9. If the call succeeds, Alice's score is updated in the LBCR of Protocol A.

5.3.5 Trying to integrate with Aave

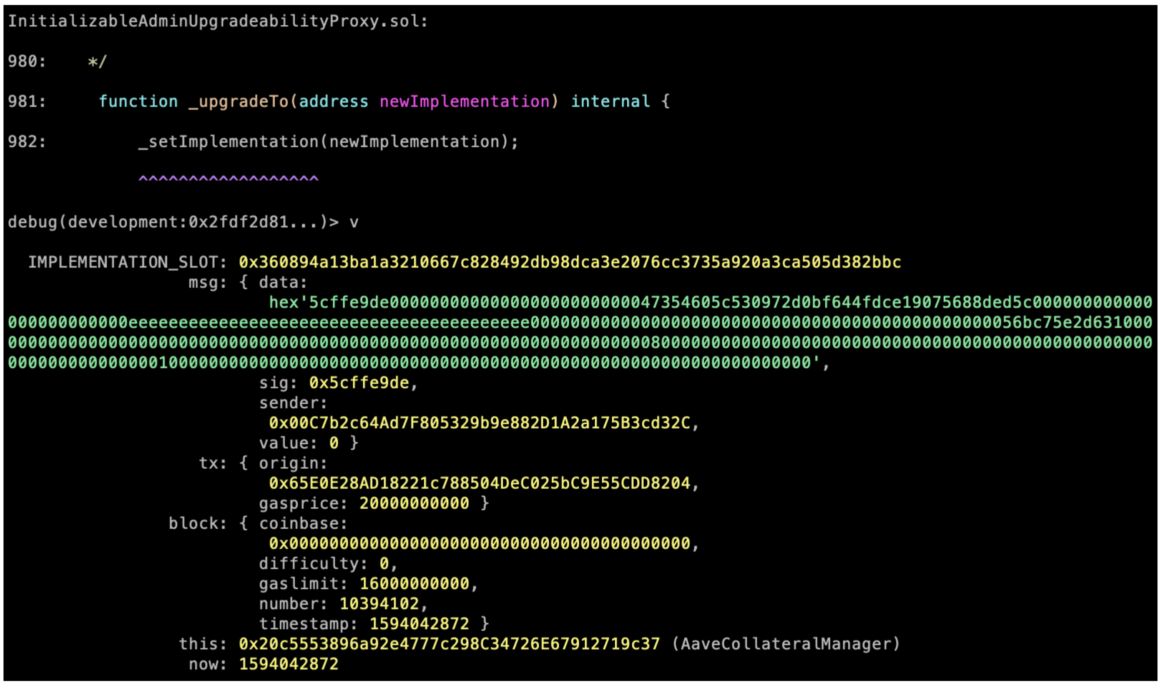
Because work on integrating Byzantic with Aave had already started with the occasion of the New York blockchain week hackathon (Section 5.2.3), the Protocol Proxy for Aave was already implemented. To make the second Byzantic implementation work with Aave, all that was left was writing an Aave CollateralManager. However, in spite of writing a very basic Collateral Manager, all state-changing transactions would revert. The Truffle Debugger was of no help, as it would often crash when stepping through the delegatecalls. When it would not crash, the fact that `delegatecalls` execute in the storage of the calling contract caused the debugger to skip large portions of code or not display any variable values.

Figure 5.4 is an example debugging attempt with Truffle. The code being executed is Aave's `InitializableAdminUpgradeabilityProxy` contract, as can be seen on the first line. This contract is Aave's upgradeability proxy, which redirects all incoming calls to the latest `LendingPool` contract. The executing context is Byzantic's `AaveCollateralManager` (penultimate line in the figure), because the code was called using `delegatecall`. Most likely, the variable `newImplementation` has no value because it does not exist in the storage context of `AaveCollateralManager`. Nonetheless, the debugger was useful in figuring out that Aave uses an upgradeability proxy, a fact enforced by comments found in Aave's smart contracts. Aave's documentation does not mention anything about upgradeability proxies [96], so we could only observe this by exploring the code itself.

Finally, after we reached out for support on Aave's Discord channel, a developer confirmed that using a `delegatecall` will not influence the protocol's storage and that the problem should be approached differently.

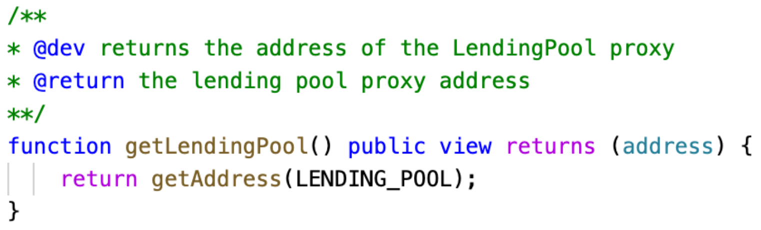
5.3.6 Challenges

Unfortunately, this solution is not technically feasible. The only way proxy functionality could be achieved in the EVM is via `delegatecall`, which does not work as needed. The Ethereum Improvement Proposal (EIP) for `delegatecall` specified the following behaviour: "CALLER and VALUE behave exactly in the callee's environment as they do in the caller's environment" [97]. The proposal makes a clear distinction between the environment of the callee and that of the caller, but specifies that they should behave identically. If `delegatecall` were implementated like in the



```
InitializableAdminUpgradeabilityProxy.sol:

980:    */

981:     function _upgradeTo(address newImplementation) internal {

982:         _setImplementation(newImplementation);

             ^^^^^^^^^^^^^^^^^^

debug(development:0x2fdf2d81...)> v

  IMPLEMENTATION_SLOT: 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc
                  msg: { data:
                          hex'5cffe9de00000000000000000000000047354605c530972d0bf644fdce19075688ded5c000000000000
000000000000eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee00000000000000000000000000000000000000000000000056bc75e2d631000
0000000000000000000000000000000000000000000000000000800000000000000000000000000000000000000000000000000000000000000
0000000000000001000000000000000000000000000000000000000000000000000000000000000000',
                         sig: 0x5cffe9de,
                         sender:
                          0x00C7b2c64Ad7F805329b9e882D1A2a175B3cd32C,
                         value: 0 }
                   tx: { origin:
                          0x65E0E28AD18221c788504DeC025bC9E55CDD8204,
                         gasprice: 20000000000 }
                block: { coinbase:
                          0x0000000000000000000000000000000000000000,
                         difficulty: 0,
                         gaslimit: 16000000000,
                         number: 10394102,
                         timestamp: 1594042872 }
                 this: 0x20c5553896a92e4777c298C34726E67912719c37 (AaveCollateralManager)
                  now: 1594042872
```

Figure 5.4: Truffle Debug failing to display the value of the variable `newImplementation`. The code being executed is Aave's `InitializableAdminUpgradeabilityProxy` contract. However, the code is not executing in the storage context of Aave's contract; it is executing in the context of the caller contract, Byzantic's `AaveCollateralManager`, because the code was called using `delegatecall`. Most likely, the memory location at which the code attempts to read `newImplementation` is uninitialized.



```
/**
* @dev returns the address of the LendingPool proxy
* @return the lending pool proxy address
**/
function getLendingPool() public view returns (address) {
    return getAddress(LENDING_POOL);
}
```

Figure 5.5: Aave source code NatSpec providing information that is missing from the documentation

EIP specification, the upgradeability proxy design pattern would not have been possible, because logic contracts would have also stored their own state - which would have been lost after upgrading to a new logic contract. It is probably for this reason that `delegatecall` was implemented differently from the EIP specification: to allow the usage of `delegatecall` for the upgradeability proxy. Nonetheless, the difference between the proposal and its implementation, coupled with the difficulty of debugging `delegatecalls`, is what delayed the development of this individual project by two weeks. Eventually, it became clear why `delegatecall` cannot be used as a regular proxy and the final architecture was designed (Chapter 5.4).

The `delegatecall` implementation makes a call to a different contract in such a way that the code of the callee contract is executed in the storage context (environment) of the caller contract. Any changes that would have been recorded in the callee contract's storage now get stored in the caller contract storage. The EVM simply overwrites the bytes in the caller contract that would have normally (i.e. using `call`) been written to in the callee contract. Unless the caller has an identical memory layout to the callee (the same variable types, declared in the same order), these changes will corrupt the memory state of the caller, resulting in transactions that revert without providing a reason. It later became apparent that transactions reverted on chained proxies because of how the Proxy Contract tried to load the target address for the `delegatecall`. The Proxy Contract was itself `delegatecalled` by the `CollateralManager`, and thus the target address in the Proxy Contract (which was used as a logic layer in this case) was being loaded from `CollateralManager` contract, which was the storage layer in this case. The target contract address that was read from memory was probably zero (i.e. the null address) or some other random address. And because that address was not a deployed contract (and thus had no fallback function), sending ETH to it would revert the transaction without a reason.

Parallel to Imperative Programming. Chaining delegatecalls can partially be compared to nesting function calls in an imperative programming language like C. In the EVM, there is no programmable global memory, as is the heap memory in C. All state is stored ”locally” in smart contracts, like in the stack frames of C. The `delegatecall` works like a function A calling function B without creating a new stack frame for function B. The code of function B is still loaded from the text segment of memory, but all reads and writes interact with the stack frame of A. Without careful memory management, the code of B could perform an out-of-bounds memory access or an invalid memory dereference, both of which are undefined behaviour in C. There is no undefined behaviour in Solidity/the EVM, but such memory mismanagement caused transactions to revert without a reason during the chained proxy call.

5.4 Final implementation

5.4.1 Changes from the previous version

The final solution simplifies the previous one by completely eliminating Collateral Manager contracts and replacing the `delegatecall` with a regular `call`. As Figure 5.6 shows, the UserProxy now directly calls contracts in target protocols. Target protocols no longer need to remove collateralisation checks from their code. Instead, they need to slightly update these checks, by multiplying whatever collateralisation ratio an action has with the aggregated User Factor (`UF`) returned by the Web Of Trust contract. To know how much liquidity was ”unlocked” for a user in a protocol, we calculate the value of `(1 - UF)` $\times$ `UTVL` , where `UTVL` is a User’s Total Value Locked in a target protocol [71].

In this implementation, **LBCR contracts are configured by protocol governance**, although they are not deployed by it. Updates to LBCR configuration create a new version of that LBCR. **Versioning is introduced** to prevent malicious updates from impacting the security of other protocols in Byzantic. If Protocol A’s governance is overtaken by a malicious actor who sets new LBCR parameters, other protocols would be affected by exaggerated reputations. Through versioning, protocols gain the option to set the default compatibility score with new LBCR versions of the other protocols. For instance, Protocol B may have set a compatibility score of 0.2 with Protocol A for the current LBCR version, but the default compatibility score for future versions is 0. This means that Protocol B governance has to manually ”trust” every Protocol A LBCR update.

Moreover, a moving average that tracks transaction volumes is added in this Byzantic implementation. It dynamically scales the curation interval in case of exceptional events where volumes rise quickly, such as a price crash.

5.4.2 Architecture

Diagram Components. We describe how each component in Figure 5.6 changed from the first implementation (Section 5.3) and why.

Web of Trust. No functionality changes.

UserProxy Factory. No functionality changes.

Alice’s UserProxy. Instead of calling the Collateral Manager contract of the target protocol, it now calls Protocol A contracts directly.

LBCR. A moving average that tracks transaction volumes is introduced, to dynamically scale the curation interval. Previous LBCR implementations would only promote or demote users one layer at a time this version replaced that strategy with

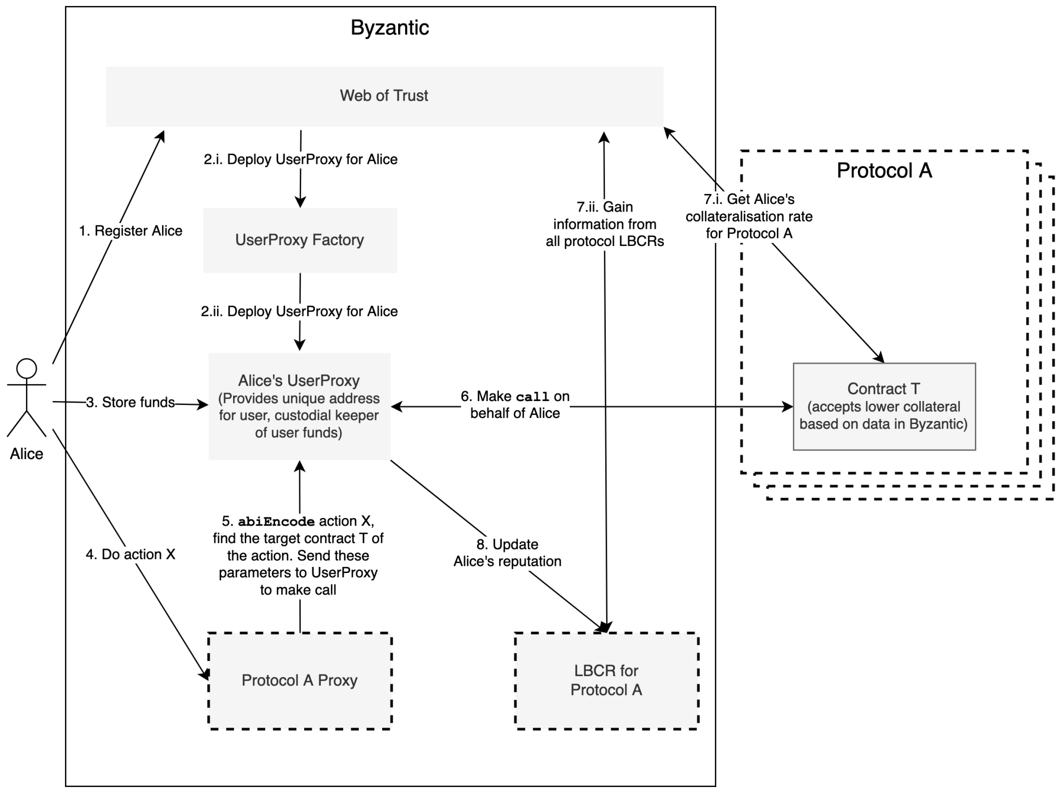

Byzantic
Web of Trust
2.i. Deploy UserProxy for Alice
UserProxy Factory
2.ii. Deploy UserProxy for Alice
1. Register Alice
Alice
3. Store funds
Alice's UserProxy (Provides unique address for user, custodial keeper of user funds)
4. Do action X
5. abiEncode action X, find the target contract T of the action. Send these parameters to UserProxy to make call
Protocol A Proxy
6. Make call on behalf of Alice
7.ii. Gain information from all protocol LBCRs
7.i. Get Alice's collateralisation rate for Protocol A
8. Update Alice's reputation
LBCR for Protocol A
Protocol A
Contract T (accepts lower collateral based on data in Byzantic)

Figure 5.6: Final system architecture of Byzantic. Contracts with dashed borders are managed by their corresponding protocol governance.

two new ones: Layer Jumping and Asymmetric Promotion.

Protocol A Proxy. No functionality changes.

Protocol A CollateralManager. This protocol was removed because `delegatecall` was not modifying the storage of Protocol A, rendering the Collateral Manager ineffective.

Protocol A. Updates collateralisation checks by multiplying the collateralisation factor with the agent discount from Byzantic.

Alice. No changes.

5.4.3 Interaction Flow

When a new agent (Alice) starts using Byzantic:

1-5 The first five steps are identical to the second implementation (Section 5.3.4).

6 Alice's UserProxy performs a low-level `call` to Contract T in Protocol A, using the `abiEncoding` from step 5.

7 Contract T of Protocol A queries the Web of Trust contract in Byzantic, which aggregates the reputation of Alice from all protocols, to retrieve her collateral reduction. If Alice does not have enough collateral even after the reduction, the call is reverted.

8 If the call succeeds, Alice's score is updated in the LBCR of Protocol A.

5.4.4 Challenges and Solutions

Gas costs of LBCR versioning. There were two options available to implement LBCR versioning. One was to use `mapping` types, from version number to the data structures of versioned parameters. The other was to use two identical data structures for every versioned parameter: one storing the current version in use, and one storing the latest version (whose values are still being updated and has not been "published"); when the latest version is "published", the new parameters are copied from one data structure to the other.

"Big O" Complexity Analysis of Gas Cost. To properly compare these two methods, their gas cost must be taken into account. The costliest part of both implementations is writing the new configuration parameters to storage. However the methods differ in that the first one always writes to uninitialized memory (that was previously zero), and the second always writes over previously initialised memory (except for the first version). The gas cost of writing a word to uninitialized storage is 20,000 gas, while writing a word to initialized storage costs 5,000 gas [98]. Given that the configuration size of a single version is identical in both protocols (the same number of layers and corresponding parameters), the number of operations to write a configuration to storage can be abstracted away. What differs is that the second method first writes to one set of data structures and then copies all of those values to another data structure. Thus, storage is written to for storing the new configuration, then it is read from and written to again, to update to a new version. Reading from storage only costs 200 gas, so in total the second method has a complexity of roughly O(10,000 gas), while the first method has one of O(20,000 gas).

Example. There are four configuration data structures: action scores, lower and upper layer score bounds and layer factors. Assuming there are 5 layers and 6 actions (inspired from SimpleLending), there are $5 \times 3 + 6 = 21$ `uint256` writes to storage, or 21 words. Currently, cost of gas is 60 gwei [99]. Thus, the cost of method 1 is $20,000 \times 21 \times 60 = 25,200,000$ gwei $\approx$ 10 dollars [100]. The cost of the second method is half of that, so about 5 dollars.

Picking an option. Given LBCR versions are likely to change very rarely and a difference of 5 dollars is not much, we decided to choose the first implementation, as it is easier to understand and debug.

5.5 General Implementation Challenges

Lack of Buidler Mainnet Forks. This missing feature was a significant obstacle to integrating with real protocols like Aave. With mainnet forks in the Buidler EVM, log statements and stack traces could have been used to more easily debug the Solidity code. The only debugging solution that offers mainnet forks is Truffle. However, the problem with the Truffle debugger was that it did not allow external (on-chain) contracts to be stepped through. An issue on the Truffle GitHub [101] suggested that including an on-chain contract in the project and compiling it to the same bytecode would help the debugger notice and step through it. However, following those instructions was unsuccessful. After looking through Truffle's code to see what is wrong, we observed that the bytecode of the on-chain contract was slightly different from the bytecode of the same contract in our project: their metadata differed (such as the project path at compilation time). But since the metadata is irrelevant to Truffle Debug's ability to step through the code, we changed the bytecode-match check such that the metadata-specific portion would be ignored. As a result, Truffle Debug is recognising the locally included on-chain contracts and, even with the occasional crashes, most transactions could be stepped through. The Truffle developers have added this missing feature in the meantime, but their solution is slow and and still crashes [102].

Chapter 6

DejaVu Design Pattern Implementation

6.1 Architecture

Components

Alice. An off-chain user that can prove ownership over a Byzantic identity.

Forward-Deployed Account. An on-chain account that Alice controls. It is the identity of this account that Alice can use by sending authenticated anonymous transactions from off-chain.

Trusted Relay. Either an single miner that relays off-chain transactions, or a network of such miners that is organised like The Onion Router in order to preserve privacy (see Section 2.3.3). In case there is a single relayer, it needs to be trusted not to expose the identity of off-chain users.

ZkIdentity. Smart contract that implements proof of preimage verification in zero knowledge by integrating with the *validator* contract generated by ZoKrates.

IdentityFactory. Simple factory contract that generates identities for the Forward-Deployed Accounts.

Identity. Smart contract that works just like the UserProxy in Byzantic (Section 5.3.3), calling DeFi protocols on behalf of the identity it corresponds to.

Protocol A. DeFi protocol that Byzantic has integrated with

Event Flow

1. The Forward-Deployed Account (FDA) controlled by Alice registers with the DejaVu Identity Factory

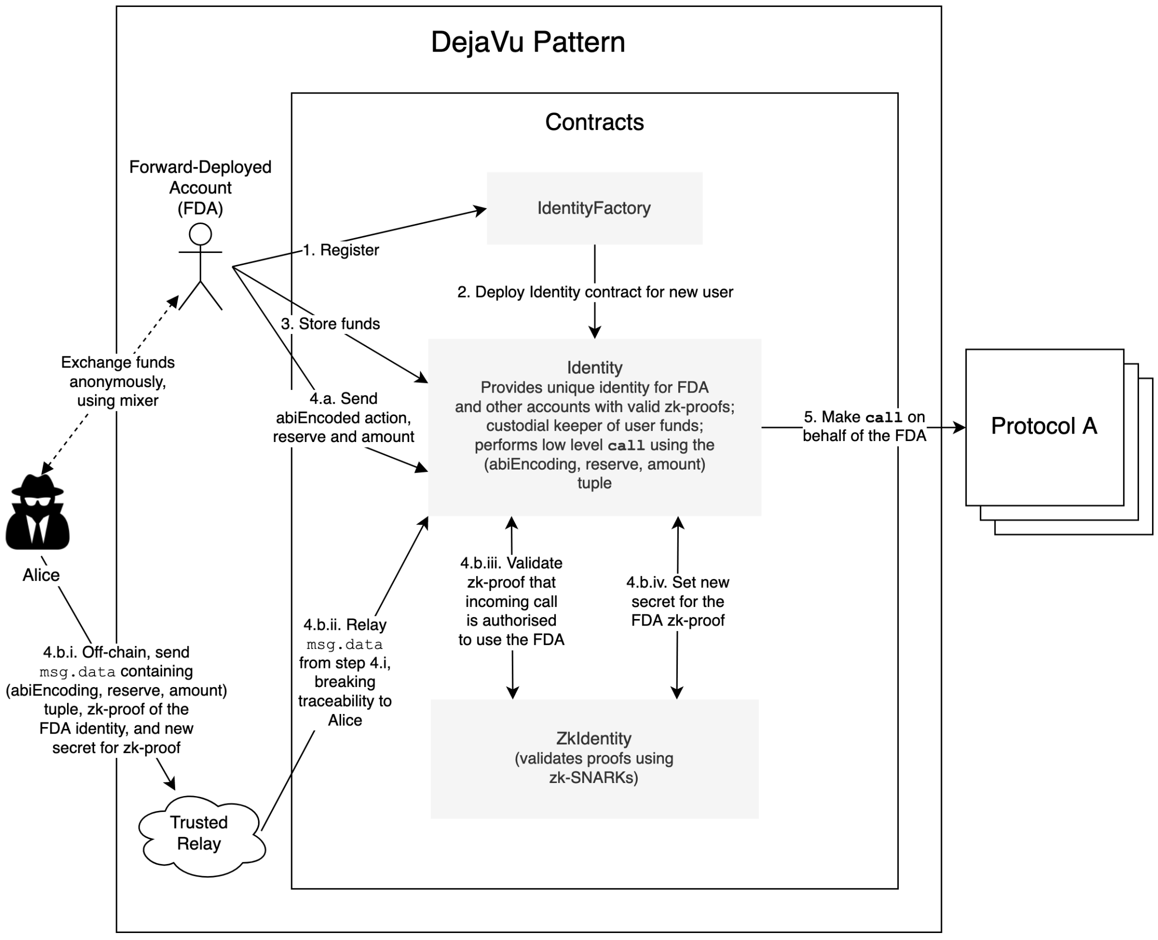

DejaVu Pattern
Contracts
Forward-Deployed Account (FDA)
IdentityFactory
1. Register
2. Deploy Identity contract for new user
3. Store funds
Identity
Provides unique identity for FDA and other accounts with valid zk-proofs; custodial keeper of user funds; performs low level call using the (abiEncoding, reserve, amount) tuple
4.a. Send abiEncoded action, reserve and amount
5. Make call on behalf of the FDA
Protocol A
Exchange funds anonymously, using mixer
Alice
4.b.i. Off-chain, send msg.data containing (abiEncoding, reserve, amount) tuple, zk-proof of the FDA identity, and new secret for zk-proof
4.b.ii. Relay msg.data from step 4.i, breaking traceability to Alice
4.b.iii. Validate zk-proof that incoming call is authorised to use the FDA
4.b.iv. Set new secret for the FDA zk-proof
ZkIdentity
(validates proofs using zk-SNARKs)
Trusted Relay

Figure 6.1: DejaVu pattern for anonymously building a transaction history. It uses both off-chain components (relays) and on-chain components (mixers and zero-knowledge proofs), providing k-anonymity guarantees.

2. The Identity Factory deploys an Identity contract for the FDA
3. The FDA deposits funds to the Identity contract. Later, these funds will be coupled with incoming data calls and forwards as ”full” transactions to DeFi protocols
4. This step can happen in two ways:
   (a) The FDA sends a transaction to the Identity contract. It does not need to provide a zk-proof, because it is the owner of the identity
   (b) i. Alice, who is off-chain, produces a zk-proof that shows she controls the Identity contract and `abiEncode`s a call for Protocol A. She sends these two pieces of data as a single call, to the Trusted Relay to forward to the Identity contract
       ii. The Trusted Relay sends Alice’s transaction using its own Ethereum account
       iii. The Identity contract calls ZkIdentity to Validate the proof. If this step fails, the transaction is reverted
       iv. The Identity contract sets a new secret for the FDA identity, using data from Alice’s call
5. The identity contract calls Protocol A using the `abiEncode`ing received in the previous step

6.2 Generating Zero-Knowledge Proofs

High-level ZoKrates programs have a `.zok` extension. To embed proof verification on-chain, the verifier needs to run the following commands.

1. Compile the `.zok` verification program to an arithmetic circuit:
   ```
   zokrates compile -i hashPreimage.zok
   ```
2. Using the generated arithmetic circuit, the verifier can run a ”trusted setup” and generate a verification key and a proving key. If there is more than one verifier, this stage is susceptible to attacks, because other verifiers need to trust the verifier running the trusted setup not to tweak the keys. In such a scenario, the setup needs to be performed via Multi-Party Computation, where one honest participant is enough to guarantee the security of the setup [103]. Fortunately, Byzantic has a single verifier, the `ZkIdentity` contract, so the trusted setup is not an issue. The following command needs to be run:
   ```
   zokrates setup
   ```
3. To generate the verifier contract that will be deployed on-chain, the following command must be run in the same directory as the verification key:
   ```
   zokrates export-verifier
   ```

4. When a user wants to generate a hash pre-image proof, the need to generate a witness (the computation that will be verified on-chain by the ZK-SNARKs). From the witness, the user can generate the proof that must be sent to the verifier smart contract. The hash pre-image does not appear in the proof due to ZK-SNARKs properties (Section 2.3.1). The following commands must be run in the same directory as the proving key:

   ```
   zokrates compute-witness -a 0 0 0 5
   ```

   ```
   zokrates generate-proof
   ```

6.3 Challenges

It is critical that users of DejaVu try to blend-in with ”the crowd” as much as possible, by transacting with very popular amounts (e.g. 0.1 ETH), at common times of the day, performing transactions that are not too sophisticated. Otherwise, they risk having their activity traced back to them.

If ZKPs may be sent to more than one verifier contract, a trusted setup must be organised, usually through a multi-party ceremony that succeeds if at least one participant is honest [104]. However, there is no need for a multi-party ceremony as there is a single verifier in DejaVu.

Chapter 7

Simple Lending Protocol

To test that Byzantic properly aggregates reputation, it was necessary to integrate it with at least two protocols. Since Truffle's debugging capabilities leave much to be desired, it was much more practical to write my own simple lending protocol (called SimpleLending) and run everything locally. By running everything locally, we could take advantage of the Buidler EVM, which sped up development compared to the previous stages of the project. Another advantage of having written my own protocol is that it can be cloned an arbitrary number of times with slightly different configurations, for testing purposes.

Given that the use case of Byzantic is collateral reduction, this solution only applies to protocols where users perform some kind of lending - a very broad subset of DeFi protocols. Examples include derivatives protocols, where users mint new assets by depositing a different kind of assets as collateral. Throughout the lifetime of derivatives, minters incur debt - which makes the minting process identical to a loan [8]. Stablecoins such as Dai can also be considered derivatives, both from a regulatory perspective [105] and from analysing their minting process. Thus, the scope of SimpleLending is broad enough to cover all collateral use cases, and its uncomplicated nature does not stand in the way of analysing how agent reputation evolves over time.

7.1 Features

7.1.1 Deposit

SimpleLending users can deposit Ether and any other ERC-20 token that stores value. No price feed oracles are used, as they are costly to use. In the context of security analysis for Byzantic, it is not relevant if prices in Byzantic match those in real markets. Section 7.2 details how exchange rates are calculated.
To keep track of deposits, SimpleLending uses a nested mapping, `userDeposits`, from the `(user, reserve)` pair to the amount. Furthermore, with every deposit, a mapping that tracks liquidity in the protocol, `reserveLiquidity`, is updated.
Deposits to not accrue interest, in an attempt to avoid implementation complexity.

The impact of this function on Byzantic reputation is positive, as it increases liquidity.

7.1.2 Computing borrowable amount

The base collateralisation rate of SimpleLending is 150%. To compute the borrowable amount from a reserve, the entire value of a user's deposits is computed, in Ether. The value of the loan is also converted to Ether, and the proportion between these two values is checked to ensure it is greater than a threshold. The threshold is calculated by multiplying the base collateralisation rate with the aggregate user factor in Byzantic. If the user is not in Byzantic, their aggregate factor is `1.0` by default.

7.1.3 Borrow

Borrows can be made in any reserve whatsoever, as long as there is enough liquidity and the user is properly collateralised. If a borrow is successful, SimpleLending updates a nested mapping, `userLoans`, from the `(user, reserve)` pair to the borrowed amount. It also decreases the `reserveLiquidity` accordingly.
Loans are interest-free, in an attempt to avoid implementation complexity.
The impact of this function on Byzantic reputation is negative, as it decreases liquidity.

7.1.4 Repay

The `repay` function is used to clear loans in a certain currency. Even if several loans were made from the same reserve, they can all be repaid at once. Users can repay arbitrary amounts, and they can do so on behalf of other users. `onBehalf` is a `repay` function parameter, and when repaying their own loans, users should assign their Byzantic UserProxy address to `onBehalf`.
A successful call to `repay` will decrease the `userLoans` of `onBehalf` by the repaid amount. It will also increase the `reserveLiquidity` by the repaid amount.
The impact of this function on Byzantic reputation is positive, as it increases liquidity.

7.1.5 Liquidate

The process of liquidation is the most complex aspect of SimpleLending. This function is meant to be called on a borrower that has become undercollateralised. The liquidator can repay some of the borrower's loans, such that the borrower becomes properly collateralised again. For doing so, the liquidator is rewarded with a liquidation discount, which means that the liquidator receives the borrower collateral in exchange for repaying the loan, at a price that is 10% better than the exchange rate in SimpleLending.
Through liquidation, SimpleLending avoids being exposed to too much default risk, liquidators make a profit by taking advantage of arbitrage, and the borrowers are punished by losing 10% of the liquidated amount. In order for liquidation to be an

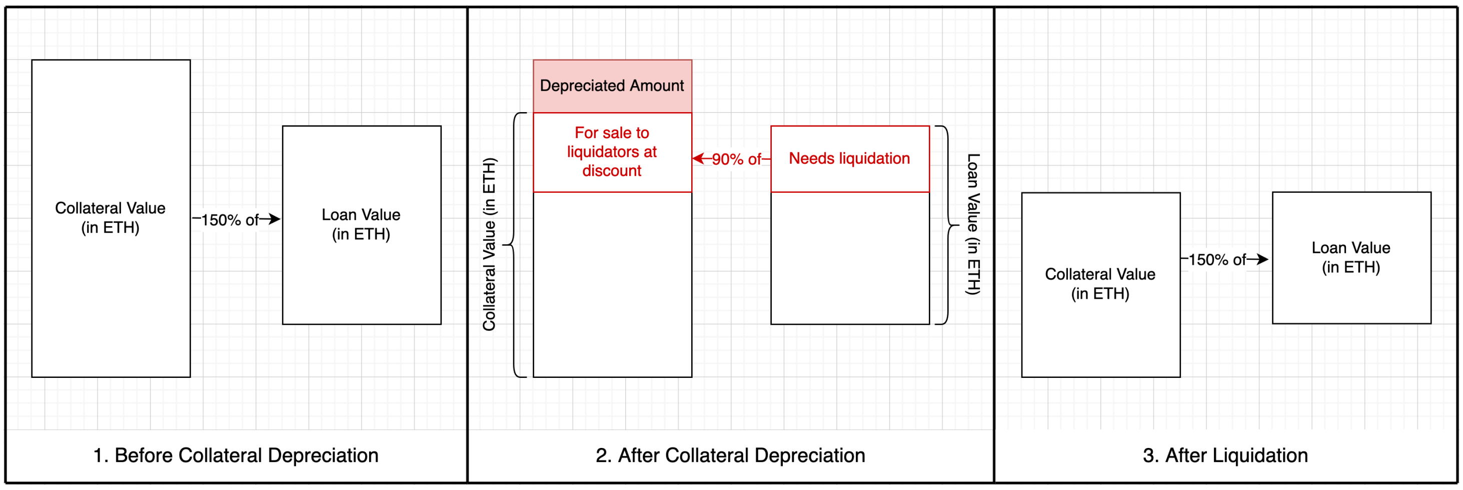

Collateral Value (in ETH)
150% of
Loan Value (in ETH)
1. Before Collateral Depreciation
Depreciated Amount
For sale to liquidators at discount
90% of
Needs liquidation
Collateral Value (in ETH)
Loan Value (in ETH)
2. After Collateral Depreciation
Collateral Value (in ETH)
150% of
Loan Value (in ETH)
3. After Liquidation

Figure 7.1: Loan liquidation process in the Simple Lending protocol. The first step illustrates the 150% ratio between values of the loan and its collateral. Then, a depreciation of the collateral assets may lead to the loan no longer being collateralised above the 150% threshold. As a result, liquidators can repay part of the loan on behalf of the borrower, in exchange for collateral at a better rate than the market's. In this example, the liquidated amount is "sold" at a 10% discount, which means that liquidators receive an additional 11.1% (100/90) of the collateral assets than they would receive on the market in exchange for the repaid loan amount.

efficient risk lowering mechanism, it is essential that the liquidation discount (10%) is lower than $1 - \frac{1}{collateralisationRatio}$. Otherwise, the liquidated will never pass the collateralisation threshold. The liquidation mechanism is not guaranteed to guard SimpleLending against sharp price drops in the value of collateral, which is why in most DeFi protocols liquidation can happen as soon as a user falls below the 150% threshold. It only becomes convenient for users to default on their loan after their collateralisation rate falls below 100%, but arbitrage is likely to occur well before that.

7.1.6 Redeem

This function can be used to withdraw deposits from SimpleLending. Users cannot redeem funds from a reserve they have not deposited to - they cannot withdraw from other users' balance. The only scenario where users are not able to withdraw the entire amount they deposited is if they were liquidated.

A successful call to `redeem` will decrease the `reserveLiquidity` and the `userDeposits` of the caller by the amount redeemed.

The impact of this function on Byzantic reputation is negative, as it decreases liquidity.

7.2 Exchange rates

The exchange value between two currencies is set based on their liquidity in SimpleLending. As liquidity is tracked using the `reserveLiquidity` mapping (Section 7.1.1), the conversion rate between asset A and asset B is:

$$\frac{reserveLiquidity[B]}{reserveLiquidity[A]} \tag{7.1}$$

The conversion rate formula was developed independently, but was afterwards compared with the approach of Uniswap, a decentralized exchange, and was found to be identical [106].
If the reserve of asset A is empty, the conversion rate returned is 2^{200}, to represent infinity. Attempts to borrow asset A would fail anyways, but returning 2^{200} is meant to act as an indication for users to smart contracts not to spend gas on a borrow transaction that will revert.

7.2.1 Example

Suppose that in SimpleLending there are 2 units of ETH and 600 units of Dai. According to the formula, the conversion rate from ETH to Dai is $\frac{600}{2} = 300$. To see if the formula obeys supply and demand laws, the price of Dai should drop if Dai liquidity increases, all else being kept the same. Thus, assume there are now 900 units of Dai in SimpleLending. The conversion rate from ETH to Dai becomes $\frac{900}{2} = 450$; using 1 ETH one can buy more Dai than before, which means Dai depreciated.

7.2.2 Challenges and Solutions

Issues with floating point numbers

Many conversions end up being smaller than 1 (for instance, Dai to ETH). This problem is made worse by ETH being stored as wei, which is 10^{18} times more fine-grained than ETH. To tackle this issue, divisions are always preceded by multiplying the numerator with 10^{25}. Instead of dividing by that offset number after the calculation is done, functions in SimpleLending return a `(number, decimals)` tuple, representing the result and the power of 10 it should be divided by ($10^{decimals}$).
In case a multiplication between numbers represented this way occurs, the fractional part of the result (the mantissa) becomes 50 digits long. In Solidity, the `uint` type stores numbers as large as 2^{256}, or approximately 10^{77}. This leaves only 27 digits for the rest of the number, which may become a problem. A potential solution is dividing the smaller number by 10^{25}, to lose as little ”information” as possible. Nonetheless, no such multiplication occurs in SimpleLending.
Another issue with the tuple representation of floating point numbers is that clients should be aware of this ”layout” when parsing the results. In JavaScript, the most common client for interacting with the Ethereum blockchain [92], the maximum safe size of numbers is 2^{53} or about 10^{15}. This is significantly smaller than the 10^{77}

in Solidity, so issues can arise when deserializing responses. JavaScript does provide the `BigInt` class for manipulating arbitrarily large numbers, but it is an additional hurdle to development. Another downside to using `BigInt` is that, like in Soldiity, fractional results are truncated. To represent a number literal as `BigInt`, the ”n” character must be appended at its end (e.g. ”10n” instead of ”10”).

”Hacking” the exchange rate

The exchange rate formula above is susceptible to transaction reordering within the same block. The vulnerability can be exploited when the miner has large reserves of a particular asset, or there is low liquidity in SimpleLending. If there is an incoming deposit transaction in ETH, the miner can short ETH, add the deposit transaction to deflate ETH price, and then repay the short position. Such an exploit can even be crafted in a single transaction with the help of flash loans, as has already happened in DeFi [107].

As a solution, Uniswap v2 are using an oracle that returns the exchange rate measured at the beginning of the block/end of the previous block. Miners are only able to manipulate this oracle if they mine consecutive blocks, which is less likely than reordering transactions. Nonetheless, manipulating price oracles could still be used as a way to increase selfish mining profits (see Section 2.7.3). If the miner fails to successfully propagate two consecutive blocks, even if they manipulate the price at the end of the one block, their arbitrage attempt could be taken advantage of by someone else. This security improvement was not used in SimpleLending, because oracles are expensive and increase solution complexity.

Chapter 8

Behavioural Analysis of DeFi Protocols

Exploring the behaviour of DeFi users is helpful in modelling the impact of Byzantic. As such, we analysed transaction data from Compound, Aave and Synthetix. Three dimensions were considered: Action Ratio, Volumes and Interaction Frequency. Based on this data, we built user profiles that enabled us to approximate how the collateral reduction offered by Byzantic evolves over time, as explained in Chapter 9. The contracts analysed are Compound's cTokens and cETH, Aave's LendingPool and Synthetix' Proxy SNX Token Contract, as shown in Table A.1.

8.1 Action Ratio and Volumes

8.1.1 Approach

We counted the occurrence of every action that users performed in 30 minute intervals. This provided us with information not only about volumes, but also about the ratio of the actions performed.
The source of the data was the Ethereum dataset in Google Cloud BigQuery, which was queried using the following SQL statement:

```
SELECT
EXTRACT(YEAR FROM block_timestamp) AS dateyear,
EXTRACT(MONTH FROM block_timestamp) AS datemonth,
EXTRACT(DAY FROM block_timestamp) AS dateday,
EXTRACT(HOUR FROM block_timestamp) AS datehour,
ROUND(EXTRACT(MINUTE FROM block_timestamp) / 30) AS dateminute,
SUBSTR(input, 0, 10) AS method_selector,
COUNT(*)
FROM ‘bigquery-public-data.crypto_ethereum.transactions‘
WHERE
DATETIME(block_timestamp) >= "2019-12-31T23:30:00"
AND DATETIME(block_timestamp) <= "2020-04-08T10:00:00"
```

```
AND
(
  to_address="contract1_address"
  OR to_address="contract2_address"
  ...
)
GROUP BY
dateyear,
datemonth,
dateday,
datehour,
dateminute,
method_selector

ORDER BY
dateyear ASC,
datemonth ASC,
dateday ASC,
datehour ASC,
dateminute ASC
```

When a function is called in a smart contract, the signature of that function is hashed using the `Keccak-256` algorithm and the first four bytes of the result are included in the `msg.data` of the transaction. The hashed function signature is the first piece of information included in the `msg.data`, which is why the `method_selector` field in the SQL statement takes a substring of the first 10 characters from the field `input` (which is the `msg.data` itself). Although the method identifier is only 4 bytes long, it is represented as 8 characters in its hexadecimal representation (a byte stores 32 `bits`, while a hexadecimal digit stores 16 `bits`). To those 8 characters, 2 additional characters are prepended: ”0x” - hence the value of 10 to the parameter of the `SUBSTR` SQL function.
While smart contracts are immutable once deployed, protocols often route transactions through proxies that decide the destination of a call a transaction time (as explained in Section 2.2). This way, protocols can deploy new contracts and re-route traffic to those, without overwriting old contracts. After such an update happens, there is often ”leftover” traffic that still goes to the old contract and can skew the results of analysis not carried out carefully. Fortunately, no such updates happened in the period shortly preceding the ”Black Thursday” crash, but Synthetix did migrate to a new contract on the 10th of May [108].

8.1.2 Results

As an example of the information gathered, Figure 8.1 illustrates user activity in Aave from the 5th to the 21st of March. The red color coding signifies borrowing or its equivalent, while yellow color coding signifies repayments. Since the proportion of repayments and liquidations increases and borrowing decreases during the price

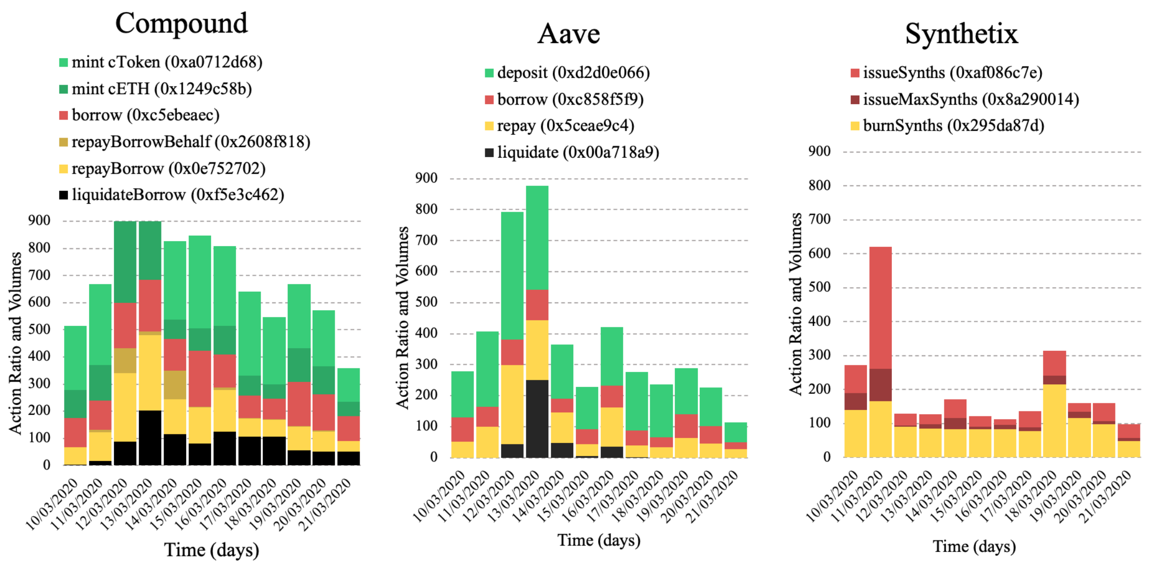


Figure 8.1: Action volumes in Compound, Aave and Synthetix before, during and after the ”Black Thursday” price crash. Not only did volumes increase overall, but the ratio of loan repayments and liquidations also increased.

crash, we consider these two actions to be the main components of calculating reputation, along with liquidation. We speculate that the increased deposit numbers are due to users providing additional collateral so as not to be liquidated, while repayments are aimed at clearing debt. Interestingly, in Synthetix, liquidation (with method identifier `0xe6203ed1`) does not occur at all, probably of its collateralisation rate of 750%, compared to 150% in Compound and Aave.

8.2 Interaction Frequency

8.2.1 Approach

The interaction frequency distribution was measured in percentiles, such that users were placed into ten ”buckets” according to their average interaction frequency. To compute the results in Figure 8.2, Google Cloud BigQuery was queried using the following SQL statement, assuming that a block is mined every 15 seconds in Ethereum:

```
SELECT
PERCENTILE_CONT(avg_time_period, 0.1) OVER() AS percentile10,
PERCENTILE_CONT(avg_time_period, 0.2) OVER() AS percentile20,
PERCENTILE_CONT(avg_time_period, 0.3) OVER() AS percentile30,
PERCENTILE_CONT(avg_time_period, 0.4) OVER() AS percentile40,
PERCENTILE_CONT(avg_time_period, 0.5) OVER() AS median,
PERCENTILE_CONT(avg_time_period, 0.6) OVER() AS percentile60,
PERCENTILE_CONT(avg_time_period, 0.7) OVER() AS percentile70,
PERCENTILE_CONT(avg_time_period, 0.8) OVER() AS percentile80,
PERCENTILE_CONT(avg_time_period, 0.9) OVER() AS percentile90
FROM (
SELECT transactions.from_address, COUNT(*) AS tos_count,
(
    MAX(transactions.block_number)
    -
    MIN(transactions.block_number)
) / COUNT(*) * 15 / (60 * 60 * 24) AS avg_time_period
FROM ‘bigquery-public-data.crypto_ethereum.transactions‘ AS transactions
WHERE
(
  transactions.to_address="contract1_address"
  OR transactions.to_address="contract2_address"
  ...
)
GROUP BY transactions.from_address )
LIMIT 1
```

The SQL subquery measures the average time between transactions, for every address interacting with a contract. It considers the distance in blocks between the first and the last transaction, which is divided by the total number of transactions originating from the caller. Then, that number is multiplied by 15, to get the average interaction time in seconds, which is then scaled to days.
A drawback of this approach is that it does not consider individuals using more than one unique address, and thus the results may be biased. A thorough analysis of linkability and traceability should be performed to ensure results are as accurate as possible [109].

Outlier Elimination

Plotting the resulting distributions helped outline that the tail of the distribution is very long - there are many users who very rarely interact with protocols. We considered this category unhelpful in modeling agent behaviour, so we decided to perform outlier elimination.

As an initial data cleansing procedure, agents who only interacted once with a protocol were no longer considered. This is because they have an interaction frequency of zero, since the distance between the first and the last transaction block is zero.

The next outliers to eliminate were those who do interact with the protocols, but do so very seldom. To decide which portion of the data are outliers, we took inspiration from the approach used in box-and-whiskers plots [110]. Such plots consider outliers all data that fall outside the interval $[Q1 - 1.5 \times IQR, Q3 + 1.5 \times IQR]$, where $Q1$ and $Q3$ are the first and third quartiles respectively (i.e. the 25th and the 75th percentiles), and IQR is the interquartile range (i.e. $Q3 - Q1$). To better suit the data we had already gathered, we only considered outliers those greater than $Q3 + 1.2 \times IQR$, so as to simply discard the 90th percentile.

8.2.2 Results

Figure 8.2 displays the distributions of interaction frequencies after eliminating ”zero-frequency” accounts, but includes the 90th percentile. Table 8.1 displays the Median Interaction Frequency (MIF) before and after outlier elimination, as well as the interaction frequency of the most active 10%.

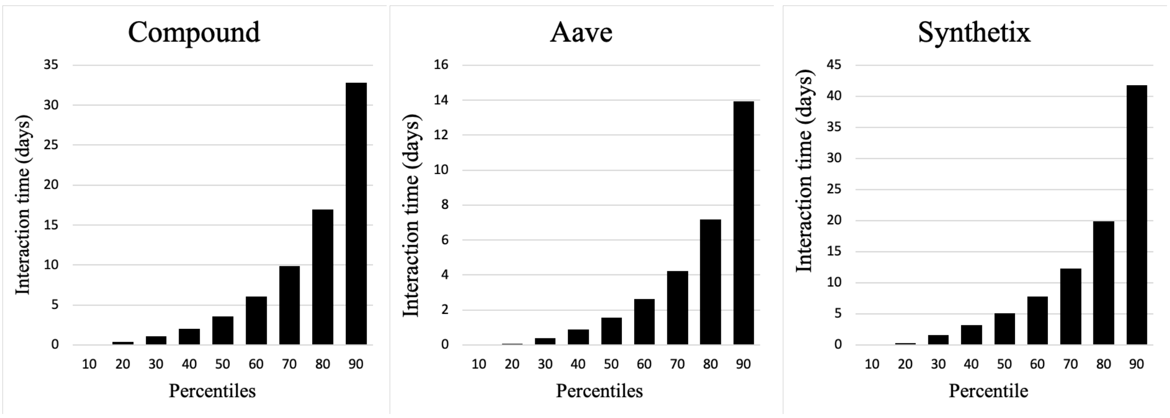


Figure 8.2: Percentiles of user interaction frequency for Compound, Aave and Synthetix.

Protocol	MFI Before OE (days)	MFI After OE (days)	10th Percentile IF (minutes)
Compound	4.8	2.81	24.12
Aave	1.57	1.23	7.12
Synthetix	4.9	4.04	8.37

Table 8.1: Summary Statistics of Interaction Frequency in Compound, Aave, Synthetix

8.3 Relationship to Market Cycle Psychology

During the crisis, users were indeed repaying their debt more than usual, reflecting the anxiety observed in traditional markets [111]. However, the ”aftershock” on the 16th of March, when the price dropped again, still caught users by surprise. This can be observed in Figure 8.1 in Aave on the 16th, in Synthetix on the 18th and in Compound on the 19th of March. It was only on the following price drop, on the 21st of March, that users seem to have been prepared for. The data shows that users were refusing to believe the price had actually crashed (i.e. they were in ”denial”). This outlines the ”fear of missing out” prevalent in DeFi due to its early stage [112].

Chapter 9

Evaluation: Economic Stress-Testing

This chapter evaluates the stability of Byzantic in the face of price shocks by simulating the system on real data.

9.1 Stress-Testing Framework

To ascertain whether DeFi protocols are vulnerable to exogenous price shocks, Gudgeon et al. [5] adapted the stress-testing methodology used by central banks, which measures the impact of ”black swan” events [113] on the financial system. By taking inspiration from their approach, we created a Python package that simulates Ethereum blockchain conditions on Byzantic, to help protocol governance securely configure their LBCR. More specifically, we analysed the effect of the ”Black Thursday” Ethereum price crash on Compound, Aave and Synthetix and present secure parameter examples. The analysed parameters are described in Sections 9.1.1 and 9.1.2.

9.1.1 LBCR Parameters

Observation. In the current implementation, there is a fixed reward per action, regardless of the funds involved. We considered this simplification good enough, since the high transaction fees in DeFi discourage users from performing the same action multiple times with small values, rather than once with the ”full” asset amount. An ideal LBCR contract would use an action reward function rather than fixed scores. Protocol governance would set a (`baseScore`, `baseTransactionAmountInETH`) pair for each action, defining the standard reward. At transaction time, the value of transacted assets is converted to ETH and the aforementioned pair is used to establish the reward score. A complication of this solution is the need to have reliable exchange rates between assets. Fortunately, Byzantic is aimed at lending protocols, all of which already need to track exchange rates.

Parameter	**Description**
Layer Factors	The factor of every layer specifies the collateral reduction (CR) users in that layer receive. The higher the layer, the higher the layer factor.
Number of layers	Assuming the highest layer factor is kept constant, this parameter only impacts the granularity in collateral reduction.
Layer Promotion Thresholds	This parameter is an array with length `numberOfLayers`, which specifies the minimum score an agent needs to have achieved in a round to be eligible for each layer.
Curation Interval	A layered-registry round ends when `curationPeriod` blocks have been mined since the round started. The LBCR brings an improvement to this concept by also considering the change in transaction volume. The LBCR computes a moving average of transaction volumes and uses the ratio between it and the current round volume to scale the curation period. Naturally, higher than usual transaction volumes shrink the curation period, to promptly adjust CRs in case of a price crash. A longer curation period allows users to perform more actions and potentially achieve better score.
Window size of the transaction volume moving average	Denotes how many previous rounds should be used to compute the moving average.
Layer Jumping (LJ)	LJ is a strategy used for LBCR layer mobility. At the end of each round, users are assigned to the highest layer whose Lower Bound is met by their score.
Asymmetric Promotion (AP)	AP is similar to LJ, but upwards mobility only happens one layer at a time, so it is a more conservative strategy. The higher the number of layers, the harder it is for agents to reach and maintain maximum reputation
Action Rewards	A mapping from `actionID` to the fixed score an agent gains or loses as a result of performing the action.
`useTimeDiscounting`	Boolean value specifying whether reputation should incorporate LBCR layer positions from previous rounds. If this parameter is set to `true`, another two parameters need to be set, expressed as percentages that add up to 100%: the weighting of last round's reputation (`w1`), and the weighting of the current layer position (`w2`). Using these two parameters, current reputation is computed as: `lastReputation` × `w1` + `currentLayerPosition` × `w2`. If `useTimeDiscounting` is `false`, the current reputation is simply: `currentLayerPosition`.

Table 9.1: LBCR Parameters

9.1.2 Web Of Trust Parameters

Parameter	Description
Compatibility Score with Protocol A	A value between 0 and 100 representing how much weighting is assigned to Protocol A in the reputation-aggregating function.
Reputation-Aggregating Function	A parameter that specifies which function should be used to aggregate reputation. It can take two values. 1. Weighted Average. Uses the compatibility score as a weighting to compute the average reputation of the user in all Byzantic protocols. Even if the aggregation does not include zero-reputation protocols (those a user has not interacted with), this approach still unfairly penalizes users for trying out a protocol, without gaining much reputation. Thus, this function encourages the concentration of transactions to a small number of protocols. The compatibility scores must add up to 100, which means that the more weighting is given to foreign protocols, the less weighting remains for the current protocol. 2. Reputation Maximiser. In this function, the compatibility score indicates the maximum amount (expressed as a percentage) of reputation in Protocol A that can be used in the current protocol. The actual percentage that will be used varies between 0 and `compatibilityScore`. The higher the reputation in protocol A, the more of its reputation will be used. If the current protocol assigns 100% compatibility score with every protocol it integrates with, then the aggregated user reputation will consist of the best reputation in any single protocol, reused in its entirety. Compatibility scores can add up to more than 100, as only the reputation forming the ”best 100” is selected.

Table 9.2: Web of Trust Parameters

9.2 Simulation Approach

We performed two kinds of simulations. One does not involve any price feeds, but relies exclusively on user profiles from the behavioural analysis of DeFi (Chapter 8) to simulate the Byzantic collateral reduction during the first quarter of 2020. The other combines the results of the first simulation with the ETH/DAI price to show the impact of the ”Black Thursday” price crash on the collateral margin of Aave,

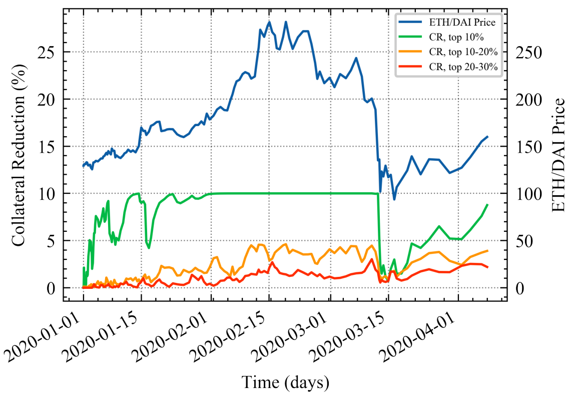


Figure 9.1: Simulation of a Byzantic configuration where only the most active users benefit from Collateral Reductions (CRs). The figure shows the reputation evolution of the top 10%, top 10-20% and top 20-30% most active users.

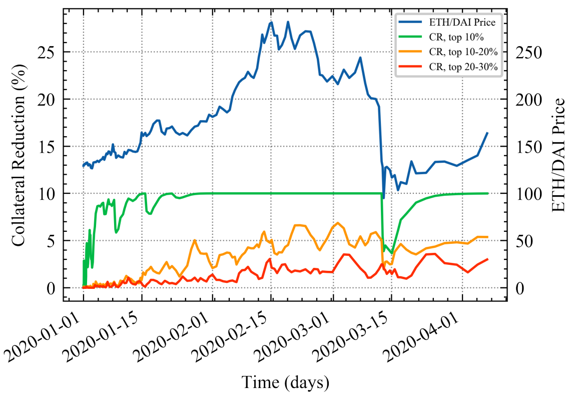


Figure 9.2: Simulation of a Byzantic configuration where the top 10-20% most active users achieve an average of 5% Collateral Reduction (CR) before the crash. During the crash, the most active users still benefit from high discounts, placing protocols at high default risk. The figure shows the reputation evolution of the top 10%, top 10-20% and top 20-30% most active users.

with and without Byzantic. We used the most active 10% of users as a reference for all simulations, who perform at least 85% of all activity in Compound, Synthetix and Aave (see Figure 8.2). This is because configuring Byzantic such that less active agents can get high discounts exposes protocols to additional default risk, as Figures 9.1 and 9.2 illustrate. The aim of this project is to maintain a low default risk at all times, and the only way to achieve this is by only allowing the most active users to benefit from discounts. We used the arbitrary value of 10% maximum collateral reduction for most simulations.

9.2.1 Simulating System Parameters

This simulation investigates the integration of Byzantic with Compound, Aave and Synthetics from the 1st of January 2020 to the 8th of April 2020, to determine secure Byzantic system parameters. Its aim is first to minimise risk during downturns and then to maximise agent utility during upturns and periods of stability. The specific actions agents perform and their frequency are simulated with user profiles from Chapter 8. For every half hour interval, we simulated 120 blocks, assuming a new block is mined every 15 seconds. The simulator was configured to use 50 users in Aave, 100 in Compound, 30 in Synthetix - proportional to the volumes we observed during the simulated time period. The results of this simulation are presented in Sections 9.3.1 and 9.3.2 and we propose examples of secure configurations for Compound, Aave and Synthetix in Section 9.3.3.

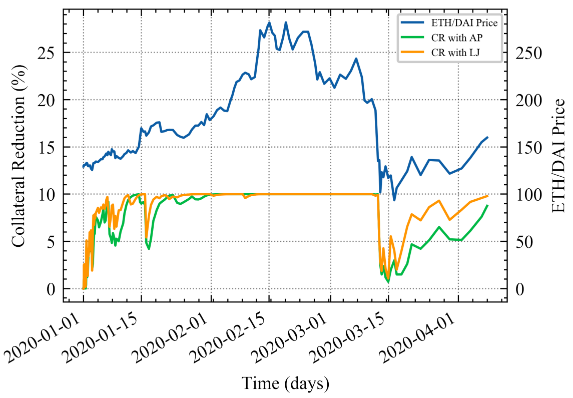


Figure 9.3: Average Collateral Reduction (CR) of the most active 10% of users in Compound using Layer Jumping (LJ) and Asymmetric Promotion (AP)

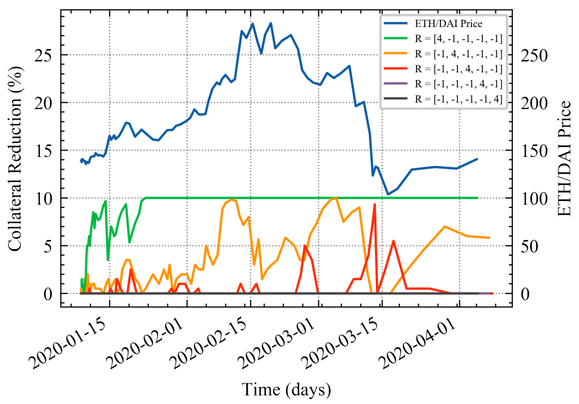


Figure 9.4: Action rewards analysis. The input values are, in order, the rewards for: deposit, borrow, repay, liquidate, flash loan. Average Collateral Reduction (CR) of the most active 10% of users in Aave, varying the Action Rewards array R

9.2.2 Simulating Collateral Requirements

Based on the estimated evolution of Byzantic collateral discounting from the first simulation, Section 9.4 details the impact of the ETH/DAI price crash on a loan of 200 DAI. The ETH/DAI pair was selected over others because in March 2020 DAI was the most commonly used cryptocurrency in DeFi [114]. The main use of over-collateralised loans is gaining more exposure to the price of the asset used as collateral [115]. For instance, traders use ETH as collateral to borrow DAI, which is then exchanged for ETH, such that they gain more exposure to ETH. Thus, the fluctuation of collateral requirements can impact trading patterns, indirectly affecting the price of DeFi assets.

9.3 System Parameter Results

9.3.1 LBCR Parameters

Layer Jumping and Asymmetric Promotion

As Figure 9.3 illustrates, AP prevents brief behavioural volatility during the crash from destabilizing reputation. LJ offers overall higher reputation at the expense of increased risk of default during a downturn. However, overall higher reputation with no additional risk can be achieved simply by lowering the Layer Promotion Thresholds.

Action Rewards

Figure 9.4 shows how different action rewards impact reputation. The input values are, in order, the rewards for: deposit, borrow, repay, liquidate, flash loan. They were chosen such that if actions were to occur with uniform probability, user score would be zero on average. The most interesting lines of the graph are the the orange and the red one, which reward borrowing and repayments respectively and are negatively correlated. It seems like borrows are a good indicator of an economic upturn, while repayments indicate downturns.

Other Parameters

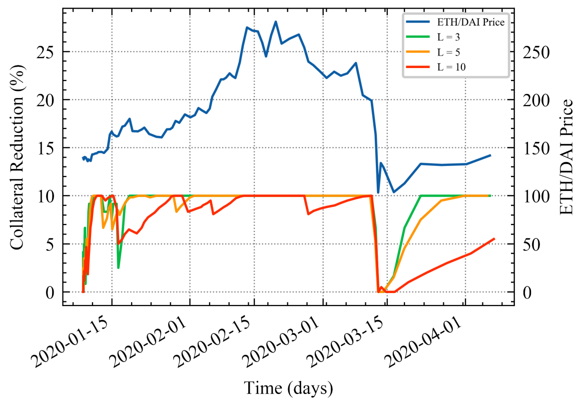


Figure 9.5: Number of layers analysis. Average Collateral Reduction (CR) of the most active 10% of users in Aave using Asymmetric Promotion (AP) and varying the number of layers L.

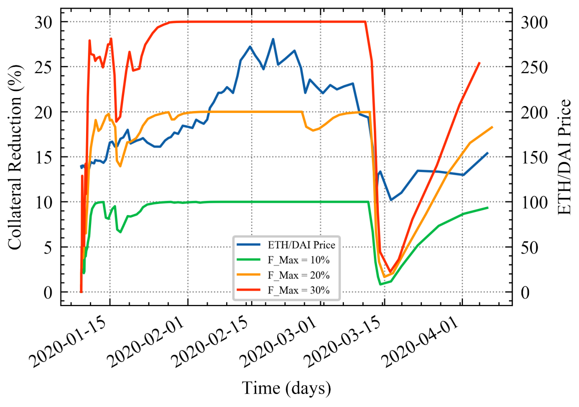


Figure 9.6: Layer factors analysis. Average Collateral Reduction (CR) of the most active 10% of users in Aave, varying the maximum and intermediate discounts.

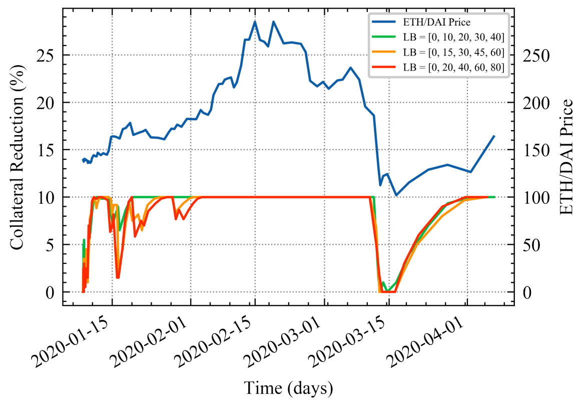


Figure 9.7: Layer promotion thresholds (LB) analysis. Average Collateral Reduction (CR) of the most active 10% of users in Aave, varying the layer promotion thresholds.

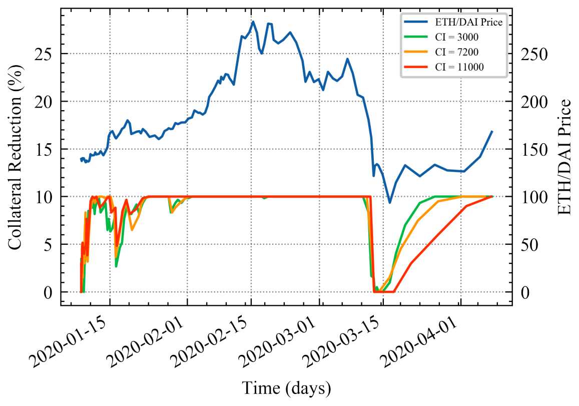


Figure 9.8: Curation interval analysis. Average Collateral Reduction (CR) of the most active 10% of users in Aave, varying the Curation Interval (CI), measured in blocks.

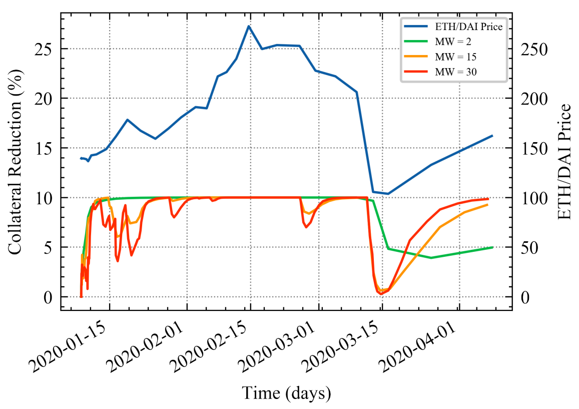


Figure 9.9: Analysis of window size for moving average. Average Collateral Reduction (CR) of the most active 10% of users in Aave, varying the window size for the moving average (MW). This value impacts how quickly the curation period ends.

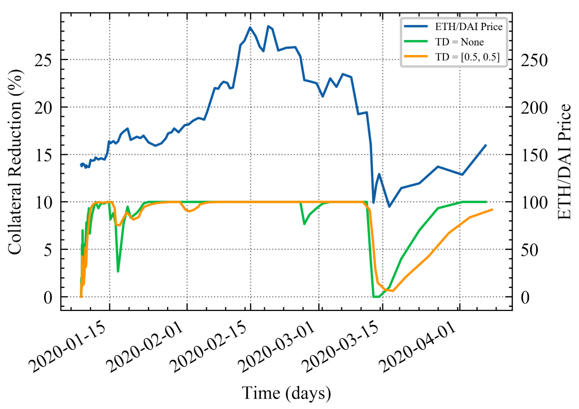


Figure 9.10: Time discounting analysis. Average Collateral Reduction (CR) of the most active 10% of users in Aave, with and without Time Discounting (TD). This value makes discount evolution smoother, slowing down the response to the price crash. To penalise undesirable behaviour from the past, the Asymmetric Promotion strategy is a better pick, since it does not slow down reputation adjustment during downturns.

9.3.2 Web of Trust Parameters

Inter-Protocol Compatibility Scores

Figures 9.11 and 9.12 show that compatibility scores (CS) should be carefully selected, based on protocol usage pattern. Moreover, the ”self” CS should always be set to 1, but it was set to 0.5 in both simulations to illustrate its impact. Aave user activity is stable, so it allows the formation of long lasting reputations. However, because the ”self” CS is set to only 0.5, agents lose utility unnecessarily if user activity in other protocols is erratic. An example of ”bursty” user activity is Synthetix, where it is important that overall high CS scores are used. Even in this case, the ”self” CS should be set to 1, as the ”compatibility maximiser” aggregation function will compute the maximum possible reputation. The ”bursty” activity pattern in Synthetix suggests that Aave should not use a high CS for it.

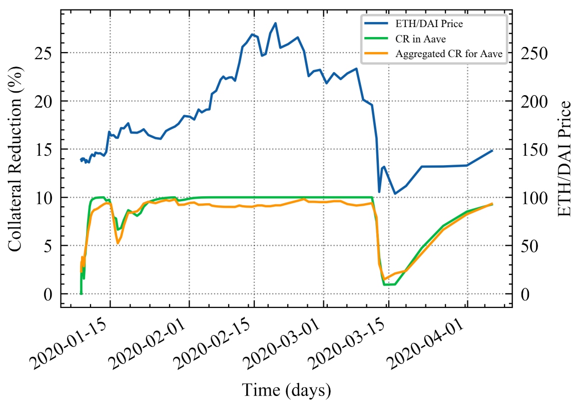


Figure 9.11: Aave compatibility scores analysis. Aggregated Average Collateral Reduction (CR) of the most active 10% of users in Aave. The aggregation function used is ”Reputation Maximiser”. The reputation in Aave is aggregated with the one in Compound and Synthetix, with compatibility scores 0.5 for its own reputation, 0.4 for Compound reputation and 0.4 for Synthetix Reputation.

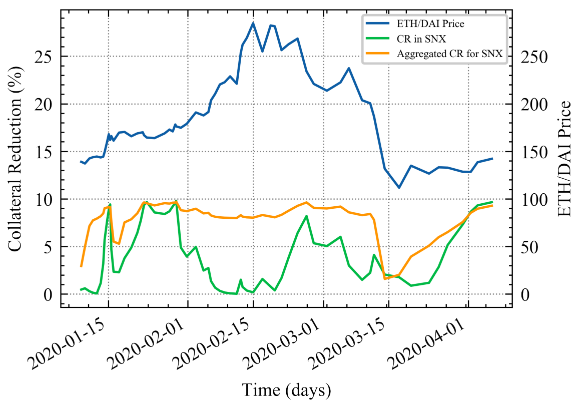


Figure 9.12: Synthetix compatibility scores analysis. Aggregated Average Collateral Reduction (CR) of the most active 10% of users in Synthetix. The aggregation function used is ”Reputation Maximiser”. The reputation in Synthetix is aggregated with the one in Compound and Aave, with compatibility scores 0.5 for its own reputation, 0.4 for Compound reputation and 0.4 for Aave Reputation.

Reputation-Aggregating Function

Simulation results show that if Asymmetric Promotion is used as a strategy for curating layers, the *reputation maximiser* aggregation function provides discounts almost as low as the *weighted average* function (Figures 9.13 and 9.14). During market upturns, the *reputation maximiser* function gives agents better utility.

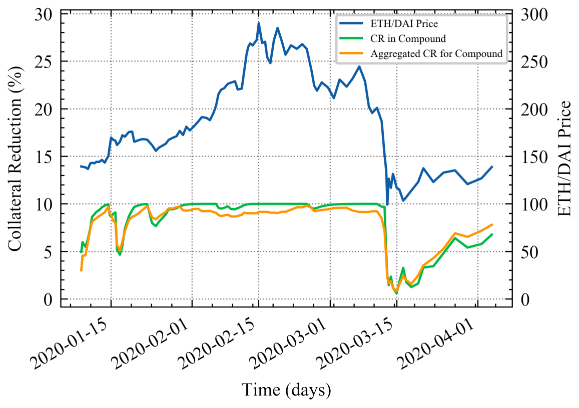


Figure 9.13: *Weighted average* aggregation function, applied to Compound. The compatibility scores used are 0.6 (self), 0.3 (Aave), 0.1 (Synthetix).

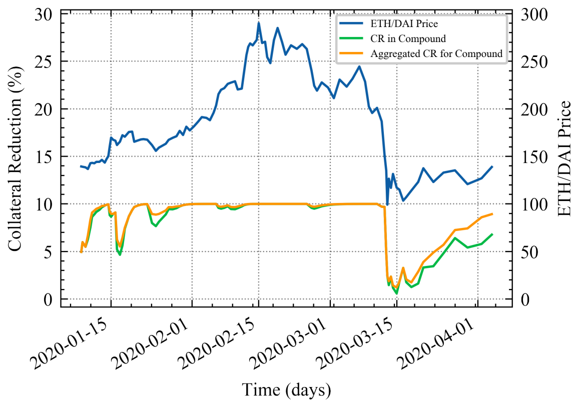


Figure 9.14: *Reputation maximiser* aggregation function, applied to Compound. The compatibility scores used are 1 (self), 0.4 (Aave), 0.3 (Synthetix).

9.3.3 Example Secure LBCR Configurations

Even if compatibility scores remain to be decided by protocols depending on the available integrations with Byzantic, we present example secure configurations in Table 9.3. These values have been picked based on simulation results from Section 9.3.1.

Protocol	Parameter	Value
Compound	Action Rewards (mint cToken, mint cETH, borrow, repayBorrowBehalf, repayBorrow, liquidate)	[2, 2, 25, -15, -15, -20]
	Number of layers	5
	Layer promotion thresholds	[0, 10, 20, 29, 37]
	Layer factors	[0, 0.03, 0.06, 0.09, 0.1]
	Curation interval	6000
	Window size for moving average	15
	Layer Jumping or Asymmetric Promotion	Asymmetric Promotion
	`useTimeDiscounting`	False
Aave	Action Rewards (deposit, borrow, repay, liquidate, flash loan)	[3, 25, -15, -20, 3]
	Number of layers	5
	Layer promotion thresholds	[0, 12, 22, 31, 40]
	Layer factors	[0, 0.03, 0.06, 0.09, 0.1]
	Curation interval	7200
	Window size for moving average	40
	Layer Jumping or Asymmetric Promotion	Asymmetric Promotion
	`useTimeDiscounting`	False
Synthetix	Action Rewards (issueMaxSynths, issueSynths, burnSynths)	[20, 20, -15]
	Number of layers	5
	Layer promotion thresholds	[0, 10, 20, 29, 37]
	Layer factors	[0, 0.03, 0.06, 0.09, 0.1]
	Curation interval	7000
	Window size for moving average	15
	Layer Jumping or Asymmetric Promotion	Asymmetric Promotion
	`useTimeDiscounting`	False

Table 9.3: Secure LBCR configurations for Compound, Aave and Synthetix

9.4 Collateral Requirements Results

Finally, using the secure Byzantic configuration for Aave from Section 9.3.3, we present a comparison between the evolution of required collateral (ETH) to borrow 200 DAI during the "Black Thursday" price crash, with and without Byzantic. The results are displayed in Figure 9.15, showing that Byzantic unlocks 30% liquidity during stable market conditions, yet collateral discounts drop to less than 3% during price crashes.

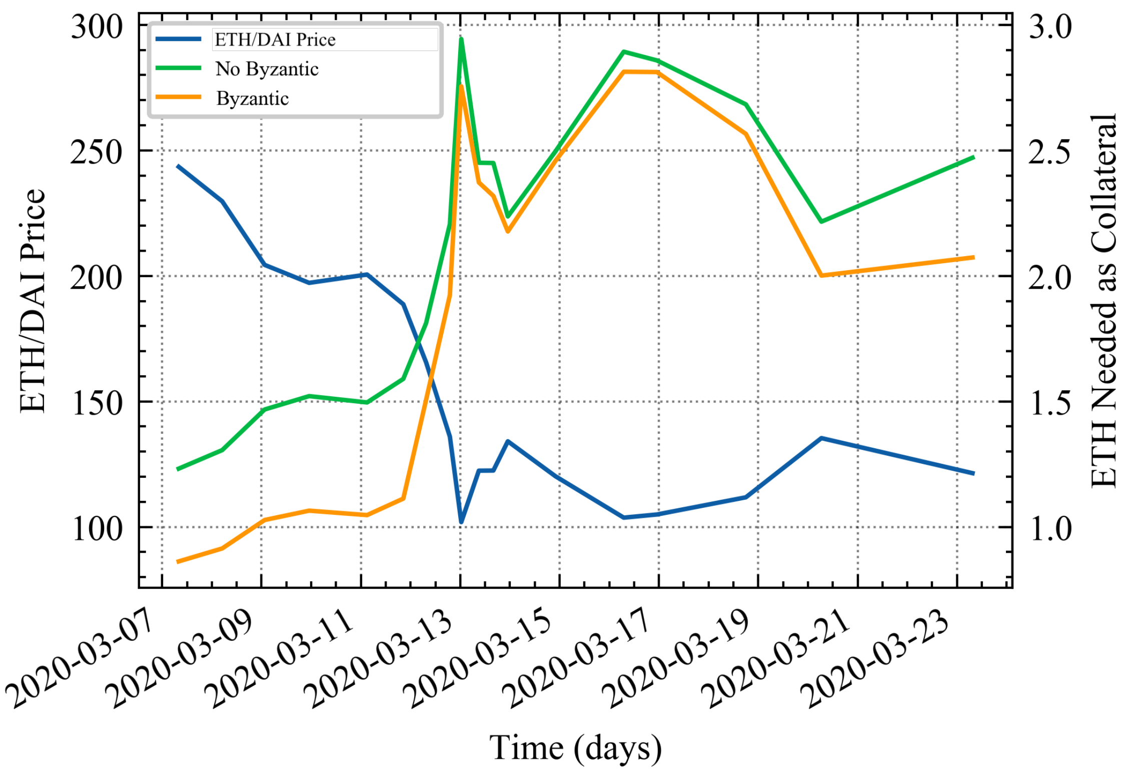


Figure 9.15: Evolution of collateral requirements for Aave users during the "Black Thursday" price crash. The blue line shows the evolution of ETH/DAI price. The green and yellow lines show the amount of ETH collateral required to borrow 200 DAI at 150% over-collateralisation rate, with and without collateral discounts from Byzantic. Before the market crash, Byzantic users achieve a collateral reduction of about 30%. During the crash, the change in their behaviour causes collateral reductions to drop to 3%. The reputation used to compute Byzantic discounts is the average of the top 10% most active agents in Aave, aggregated with reputation from Compound and Synthetix as well. There are more data points during the crash because transaction volumes increased relative to the moving average and this downscaled the curation interval.

Chapter 10

Evaluation: Solidity Implementation

This chapter evaluates the security of Byzantic using unit testing, a generic Lending Protocol implementation and two static analysers. Anonymity is evaluated using unit testing and by assuming the underlying components of the DejaVu pattern work as desired.

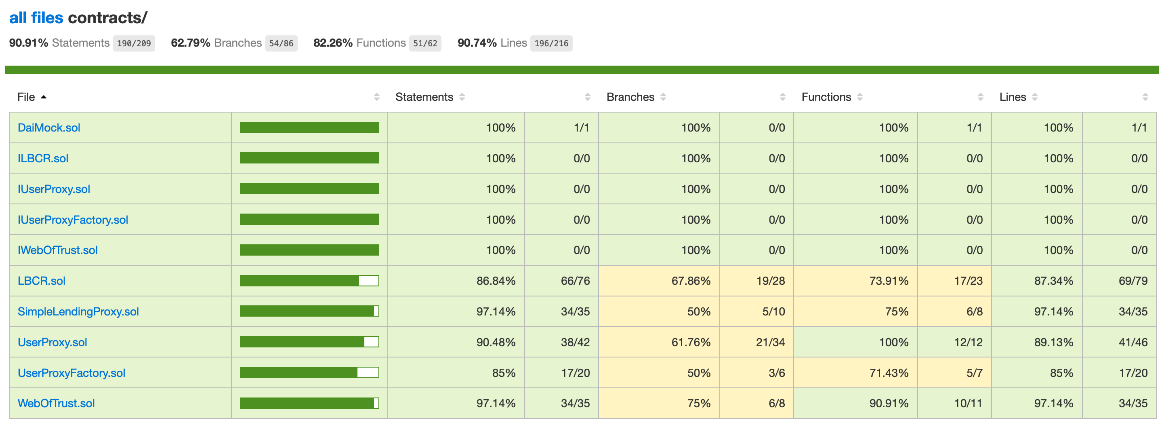
10.1 Testing

Testing was performed on two ”instances” of SimpleLending, such that reputation could be aggregated. For each of the two SimpleLending protocols, there was a deployment of:

- the SimpleLending core contract (with features such as borrow, liquidate)
- an LBCR configuration
- the SimpleLendingProxy contract (Protocol A Proxy in Figure 5.6), used to update the LBCR after each action

Two of the twenty accounts in the default configuration of the Buidler EVM were used. To initialize SimpleLending, one of the accounts would deposit assets such that their exchange rate would be at the desired value. The assets used were ETH and DaiMock, a simple implementation of an ERC-20 token.
Five integration tests were used, that encompass the entire functionality of Byzantic and SimpleLending. The tests check:

1. Depositing to SimpleLending through Byzantic
2. Depositing, borrowing, and repaying to SimpleLending through Byzantic
3. Depositing, borrowing, and being liquidated by a non-Byzantic agent
4. Depositing, borrowing, and being liquidated by a Byzantic agent
5. Depositing to SimpleLending and SimpleLendingTwo in different amounts, to gain different reputation and aggregate it



all files contracts/

90.91% Statements 190/209 62.79% Branches 54/86 82.26% Functions 51/62 90.74% Lines 196/216

File		Statements		Branches		Functions		Lines	
DaiMock.sol		100%	1/1	100%	0/0	100%	1/1	100%	1/1
ILBCR.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
IUserProxy.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
IUserProxyFactory.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
IWebOfTrust.sol		100%	0/0	100%	0/0	100%	0/0	100%	0/0
LBCR.sol		86.84%	66/76	67.86%	19/28	73.91%	17/23	87.34%	69/79
SimpleLendingProxy.sol		97.14%	34/35	50%	5/10	75%	6/8	97.14%	34/35
UserProxy.sol		90.48%	38/42	61.76%	21/34	100%	12/12	89.13%	41/46
UserProxyFactory.sol		85%	17/20	50%	3/6	71.43%	5/7	85%	17/20
WebOfTrust.sol		97.14%	34/35	75%	6/8	90.91%	10/11	97.14%	34/35

Figure 10.1: Coverage report generated by `solidity-coverage`

Measuring Test Coverage. In the remainder of this section we present details about how test coverage was determined.

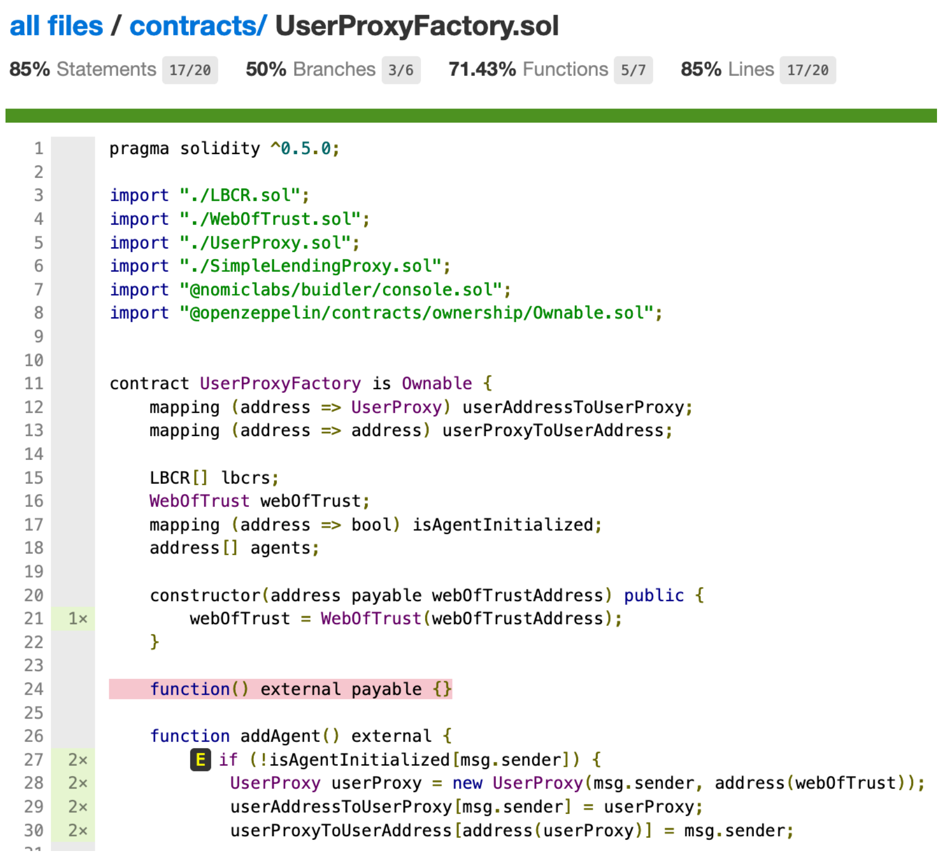
To measure coverage, the tool `solidity-coverage` came in very handy. The tool's development was still very actively ongoing during this thesis project [116], and several bugs are still unresolved issues on GitHub. The novelty of `solidity-coverage` shows how lacking blockchain tooling still is. The tool neatly generates a visual interface that displays code not covered directly by calls from external clients (i.e. JavaScript). So, a drawback is that the SimpleLending functions called by the SimpleLendingProxy contract were not considered covered, because those functions were not called directly from the TypeScript tests.

Fortunately, `solidity-coverage` already has a Buidler plugin, so changing existing tests was not necessary. The plugin adds a Buidler "task" to the existing suite (everything Buidler runs, such as `compile`, `test`, is called a task). Besides the plugin, all that was needed was adding a custom network named `coverage` in the Buidler configuration. `solidity-coverage` launches its own Ganache server to measure coverage of the instrumented smart contracts, and the custom server points Builder to that Ganache instance. Instrumentation happens at compile-time, because EVM bytecode is not expressive enough to infer line numbers. This was another issue that came up while debugging the Truffle debugger (Section 5.5). The command to run coverage analysis simply used the new network added to the Buidler config:

```
npx buidler coverage --network coverage
```

Compile-time instrumentation is a great choice for code analysis tools on Ethereum. All source code is public. Smart contract code can be found on Etherscan), but more often than not the GitHub repositories of DeFi tools are public too. This means that anyone can run coverage analysis of smart contracts using the test suite created by smart-contract authors. This is in contrast with traditional software engineering, where code is compiled to binaries, which are difficult to reverse engineer. On Ethereum and blockchains in general, reverse engineering is not necessary.

Because everything is publicly accessible anyways, OpenZeppelin, a firm creating standards and tools for Ethereum, have taken even more steps towards transparency. They made coverage analysis, which runs as part of their continuous integration pipeline, public too [117].


all files / contracts/ UserProxyFactory.sol

85% Statements 17/20 50% Branches 3/6 71.43% Functions 5/7 85% Lines 17/20

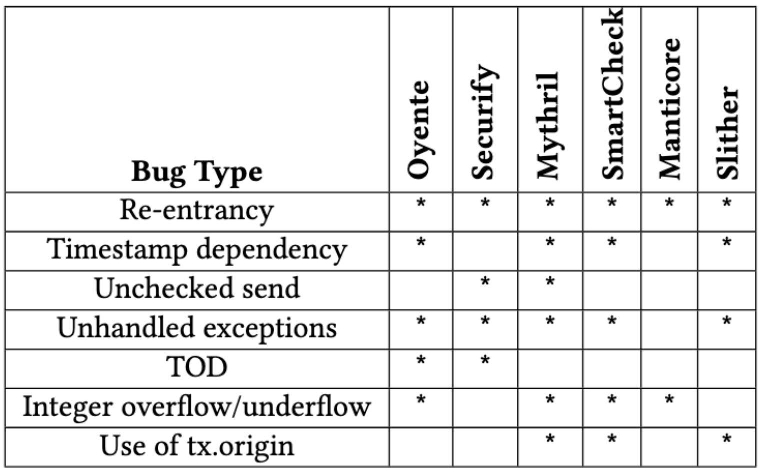
```
pragma solidity ^0.5.0;

import "./LBCR.sol";
import "./WebOfTrust.sol";
import "./UserProxy.sol";
import "./SimpleLendingProxy.sol";
import "@nomiclabs/buidler/console.sol";
import "@openzeppelin/contracts/ownership/Ownable.sol";


contract UserProxyFactory is Ownable {
    mapping (address => UserProxy) userAddressToUserProxy;
    mapping (address => address) userProxyToUserAddress;

    LBCR[] lbcrs;
    WebOfTrust webOfTrust;
    mapping (address => bool) isAgentInitialized;
    address[] agents;

    constructor(address payable webOfTrustAddress) public {
1×      webOfTrust = WebOfTrust(webOfTrustAddress);
    }

    function() external payable {}

    function addAgent() external {
2×    E if (!isAgentInitialized[msg.sender]) {
2×          UserProxy userProxy = new UserProxy(msg.sender, address(webOfTrust));
2×          userAddressToUserProxy[msg.sender] = userProxy;
2×          userProxyToUserAddress[address(userProxy)] = msg.sender;
```

Figure 10.2: Coverage analysis generated by `solidity-coverage`. It shows how many times each line has been covered and highlights lines of code that were not directly called from JavaScript.

As Figure 10.2 shows, tests cover over 90% of the code in the project. The code not covered typically consists of getter functions or fallback functions. In fact, not covering fallback functions can be considered a good thing, showing that every call to the contract was successful.

10.2 Static Analysis

Byzantic needs strong security guarantees in order to safely reduce collateral. Static analysis usually has no false negatives, so if an implementation vulnerability (as opposed to a logic vulnerability) exists, it will be reported. Static analysis can uncover vulnerabilities such as reentrancy, division by zero, and improper use of low-level calls.



Bug Type	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	*	*	*	*	*	*
Timestamp dependency	*		*	*		*
Unchecked send		*	*			
Unhandled exceptions	*	*	*	*		*
TOD	*	*				
Integer overflow/underflow	*		*	*	*	
Use of tx.origin			*	*		*

Figure 10.3: Bug types tracked by Ghaleb and Pattabiraman 2020 [118]: ”*” means that the tool can detect the bug type

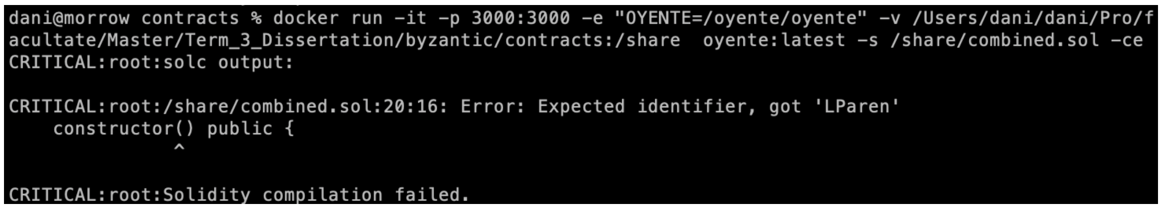
Although static analysis has no false negatives in theory, in practice it has been shown that no Solidity static analyser is free of false negatives [118]. To have as few false negatives as possible, Byzantic code was analysed by tools with complementary false negative bug categories.
Ghaleb and Pattabiraman 2020 [118] have evaluated smart contracts injected with bugs, using the most popular static analysis tools: Oyente [119], Securify [120], Mythril [121], SmartCheck [122], Manticore [123], Slither [124]. Slither was used by default in this project, because it is actively maintained and setup is straightforward. The tool is also recommended by ConsenSys [10], one of the most popular companies in the DeFi space. However, Slither does not detect Transaction Order Dependency (TOD) bugs, which can be exploited by reordering transactions within the same block (see Section 7.2.2). To compensate for Slither’s inability to detect TOD, we had to pick between the two tools that do detect it: Oyente and Securify2 (see Figure 10.3). Securify2 was considered over Securify, because the latter has been deprecated. TOD bugs were considered more important than integer overflow/underflow, as the latter is prevent by the usage of the SafeMath library in Byzantic.
Oyente and Securify2 are able to identify the TOD bugs because they use symbolic execution: if two branches in the symbolic execution tree cover the same code, but the transaction amount in one branch differs from the transaction amount in the other only because of transaction order, then a TOD bug was found.

10.2.1 Oyente

Since Oyente ticked more boxes than Securify2 for bug types Slither cannot find (Figure 10.3), it seemed the natural choice. Upon running the example analysis, a warning was issued:

```
WARNING:root:You are using solc version 0.4.21, The latest supported
version is 0.4.19
```

The Solidity version of this project is `0.5.x`, so the warning renders Oyente unusable. After trying to run the latest version of Oyente from an Ubuntu image in Docker, the compiler failed to parse the code, confirming that this tool cannot be used in Byzantic.



```
dani@morrow contracts % docker run -it -p 3000:3000 -e "OYENTE=/oyente/oyente" -v /Users/dani/dani/Pro/f
acultate/Master/Term_3_Dissertation/byzantic/contracts:/share  oyente:latest -s /share/combined.sol -ce
CRITICAL:root:solc output:

CRITICAL:root:/share/combined.sol:20:16: Error: Expected identifier, got 'LParen'
    constructor() public {
               ^

CRITICAL:root:Solidity compilation failed.
```

Figure 10.4: Error thrown when running Oyente on Byzantic, confirming that the tool is incompatible with Solidity versions newer than `0.4.x`.

10.2.2 Securify2

Since Oyente could not be used to identify TOD bugs, Securify2 was the only tool left. Securify2 is, in fact, better at finding true positives than Oyente and Mythril [120], which also use symbolic execution. The increased accuracy of Securify2 is due to its better symbolic state coverage.
The setup was very similar to Oyente, in that it is easiest to run from a Docker image. Building the Docker image is quite time consuming (takes about 5 minutes), and this is another reason why Slither is much better for performing quick analysis.
The tool was run using:

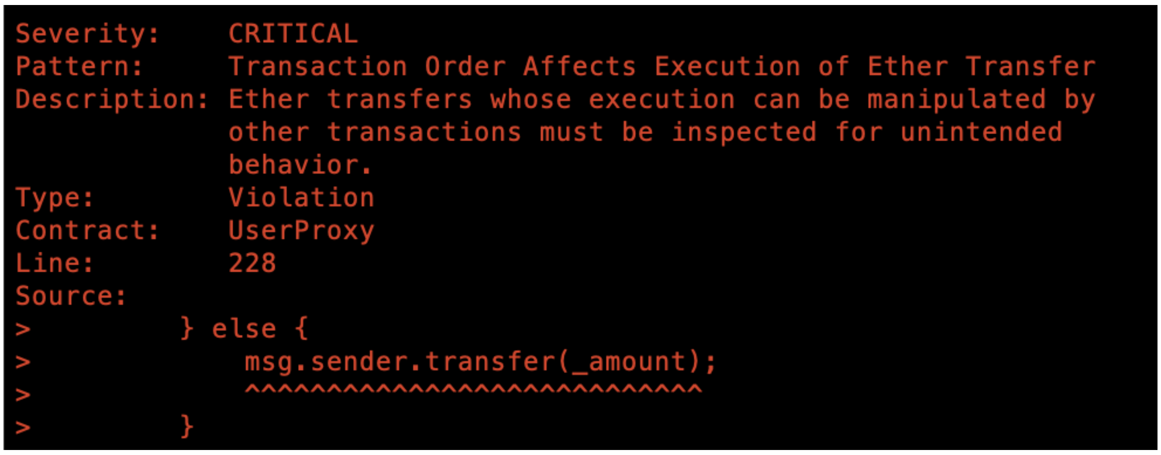
```
docker run -it -v /path/to/Byzantic/contracts:/share securify
/share/combined.sol
```

The flags in the command above are:

- `-it`: Short for `--interactive` and `--tty`, which run a terminal session within the container and keep I/O interactive. It was useful because it ensured Securify2 output was printed to `stdout`
- `-v`: Short for `--mount`, it mounts a volume named `share` in the container, whose memory is read from and written to the directory at `/path/to/Byzantic/contracts`. In this case, it is the way Byzantic contracts are passed to the Securify2 executable at the end of the command: `securify /share/combined.sol`

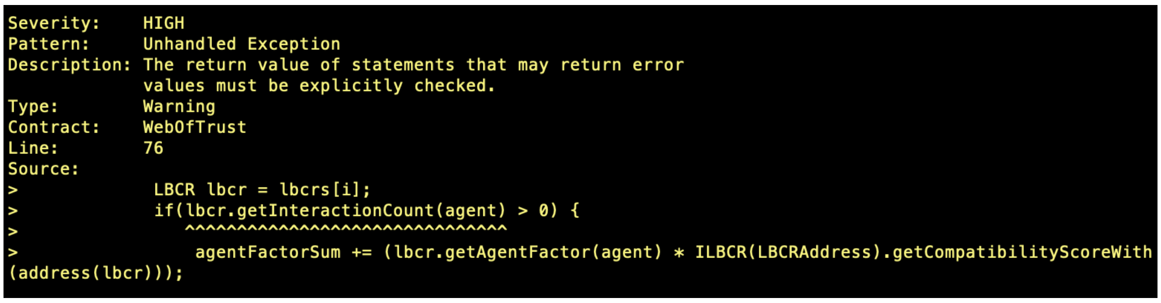
Results

Indeed, Securify2 reported potential TOD bugs (Figure 10.5), as well as other bugs (Figure 10.6). The TOD warning is indeed a bug, as the exchange rate in SimpleLending can be manipulated based on when Ether is sent to its contract. However, this is a bug in SimpleLending which has been discussed in 7.2.2, even if the line that triggers it is in Byzantic.



```
Severity:    CRITICAL
Pattern:     Transaction Order Affects Execution of Ether Transfer
Description: Ether transfers whose execution can be manipulated by
             other transactions must be inspected for unintended
             behavior.
Type:        Violation
Contract:    UserProxy
Line:        228
Source:
>          } else {
>            msg.sender.transfer(_amount);
>            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>          }
```

Figure 10.5: Critical-severity warning produced by Securify2



```
Severity:    HIGH
Pattern:     Unhandled Exception
Description: The return value of statements that may return error
             values must be explicitly checked.
Type:        Warning
Contract:    WebOfTrust
Line:        76
Source:
>             LBCR lbcr = lbcrs[i];
>             if(lbcr.getInteractionCount(agent) > 0) {
>                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>                 agentFactorSum += (lbcr.getAgentFactor(agent) * ILBCR(LBCRAddress).getCompatibilityScoreWith
(address(lbcr)));
```

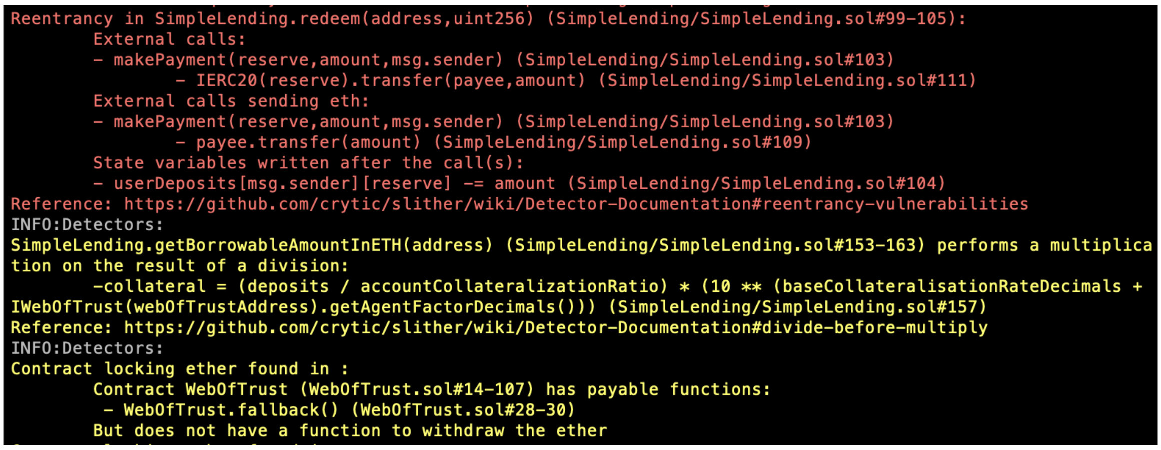
Figure 10.6: High-severity warning produced by Securify2

Challenges

Securify2 runs `solc` with a very strict value for the `allow-paths` flag: ”/”. Since most contracts import other contract in order to compile, especially npm-imported contracts, which are in an entirely different directory, the only solution was to combine all Byzantic contracts in a single contract.
To merge all contracts, sol-merger was an option. However, files had to be manually appended to a new contract from the command line, instead of the tool parsing the files for imports and merging them ”intelligently”. As a result, the contracts were merged manually.

10.2.3 Slither

Slither also produced warnings we had not noticed in the code (Figure 10.7), and helped remove vulnerabilities from Byzantic.



```
Reentrancy in SimpleLending.redeem(address,uint256) (SimpleLending/SimpleLending.sol#99-105):
        External calls:
        - makePayment(reserve,amount,msg.sender) (SimpleLending/SimpleLending.sol#103)
                - IERC20(reserve).transfer(payee,amount) (SimpleLending/SimpleLending.sol#111)
        External calls sending eth:
        - makePayment(reserve,amount,msg.sender) (SimpleLending/SimpleLending.sol#103)
                - payee.transfer(amount) (SimpleLending/SimpleLending.sol#109)
        State variables written after the call(s):
        - userDeposits[msg.sender][reserve] -= amount (SimpleLending/SimpleLending.sol#104)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
SimpleLending.getBorrowableAmountInETH(address) (SimpleLending/SimpleLending.sol#153-163) performs a multiplica
tion on the result of a division:
        -collateral = (deposits / accountCollateralizationRatio) * (10 ** (baseCollateralisationRateDecimals +
IWebOfTrust(webOfTrustAddress).getAgentFactorDecimals())) (SimpleLending/SimpleLending.sol#157)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Contract locking ether found in :
        Contract WebOfTrust (WebOfTrust.sol#14-107) has payable functions:
         - WebOfTrust.fallback() (WebOfTrust.sol#28-30)
        But does not have a function to withdraw the ether
```

Figure 10.7: Extract from static analysis report generated by `slither`

10.3 Gas Costs

To measure gas costs, we used the `eth-gas-reporter` plugin, which has a Buidler integration. After installation, all that needs to be done is including the library in the project configuration. The results of running `eth-gas-reporter` on a `deposit` call to Simple Lending are visible in Figure 10.8, and are as follows, ignoring the set up calls.

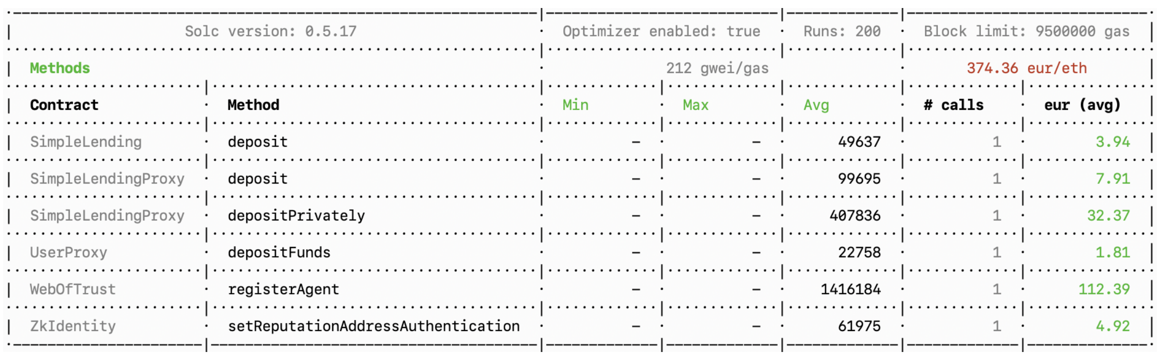
- Direct call to Simple Lending: **49637 gas**, or 3.93 Euro.
- Call to Simple Lending through Byzantic: **122453 gas** (22758 + 99695), or 9.7 Euro.
- Call to Simple Lending through Byzantic and DejaVu: **430594 gas** (22758 + 407836), or 34.17 Euro.

Thus, compared with a direct call to Simple Lending, using Byzantic costs 247% more gas. Using both Byzantic and DejaVu costs 867% more gas.

10.4 Challenges and Solutions

There are three challenges that have no immediate solution:

- Testing is slower than in non-blockchain programming languages. Partly, it is because the `solc` compiler is slower. However, the blockchain itself also has lower throughput than traditional computer memory or even centralised networked systems.
- `solidity-coverage` is both slow and inaccurate. It is slow because it does not seem to use cache: it downloads the solidity compiler and recompiles the contracts every time it is run. It is inaccurate because it does not track how calls to smart contracts ”propagate” to other smart contracts, so every smart contract function needs to be called directly from the off-chain client.



```
·------------------------------------------------------|---------------------------|-------------|-----------------------------·
|                  Solc version: 0.5.17                 ·  Optimizer enabled: true  ·  Runs: 200  ·  Block limit: 9500000 gas  │
························································|···························|·············|······························
|  Methods                                              ·               212 gwei/gas              ·       374.36 eur/eth        │
·····················|··································|·············|·············|·············|··············|···············
|  Contract          ·  Method                          ·  Min        ·  Max        ·  Avg        ·  # calls     ·  eur (avg)   │
·····················|··································|·············|·············|·············|··············|···············
|  SimpleLending     ·  deposit                         ·          -  ·          -  ·      49637  ·           1  ·        3.94  │
·····················|··································|·············|·············|·············|··············|···············
|  SimpleLendingProxy  ·  deposit                       ·          -  ·          -  ·      99695  ·           1  ·        7.91  │
·····················|··································|·············|·············|·············|··············|···············
|  SimpleLendingProxy  ·  depositPrivately              ·          -  ·          -  ·     407836  ·           1  ·       32.37  │
·····················|··································|·············|·············|·············|··············|···············
|  UserProxy         ·  depositFunds                    ·          -  ·          -  ·      22758  ·           1  ·        1.81  │
·····················|··································|·············|·············|·············|··············|···············
|  WebOfTrust        ·  registerAgent                   ·          -  ·          -  ·    1416184  ·           1  ·      112.39  │
·····················|··································|·············|·············|·············|··············|···············
|  ZkIdentity        ·  setReputationAddressAuthentication  ·      -  ·          -  ·      61975  ·           1  ·        4.92  │
·--------------------|----------------------------------|-------------|-------------|-------------|--------------|--------------·
```

Figure 10.8: Gas cost of depositing to the Simple Lending protocol directly (`SimpleLending: deposit`), through Byzantic (`UserProxy: depositFunds`, `SimpleLendingProxt: deposit`), and through DejaVu and Byzantic (`UserProxy: depositFunds`, `SimpleLendingProxy: depositPrivately`).

- Static analysis, although safer than dynamic analysis, yields many false positives that slow down development. The false positives are due to over-approximations of the smart contract runtime, made to keep the tools time-efficient.

Hard to Test Features. Furthermore, we detail challenges difficulties encountered when testing certain features.

The most challenging features to test were reputation aggregation in Byzantic and liquidation in SimpleLending. The former was greatly aided by the ability to clone SimpleLending, but developing the protocol in itself was no simple feat. The latter was achieved by using the core property of liquidity: when liquidity is low, even transacting a small amount will significantly change asset price [125]. Steps to test liquidation:

1. Agent A ”initialises” SimpleLending with only 3 ETH and 684 newly minted DaiMock. This sets the exchange rate at 228 DaiMock per ETH, while ensuring low liquidity.

2. Agent B deposits 2 ETH (to act as collateral) and takes out a loan of 150 DaiMock.

3. Agent A redeems 50 DaiMock, causing a drop in ETH-DAI exchange rate. There are 5 ETH in SimpleLending (3 from Agent A, 2 from Agent B), and 484 DaiMock (684 minus the 150 loan, minus the 50 redeem). The exchange rate is now 96.8 DaiMock per ETH, meaning that Agent B’s debt is now below the 150% collateralisation rate, at 129%. It is still not convenient for Agent B to default on their loan.

4. Agent B liquidates 46 DaiMock of Agent A’s debt in exchange of 0.52 ETH (bought at the rate 96.8 with a 10% liquidation bonus). Agent A is now left with 104 MockDai loaned and 1.48 ETH as collateral. The exchange

rate is 106 DaiMock per ETH, and Agent B's collateralisation rate becomes $106 \times 1.48/104 = 150\%$.

Chapter 11

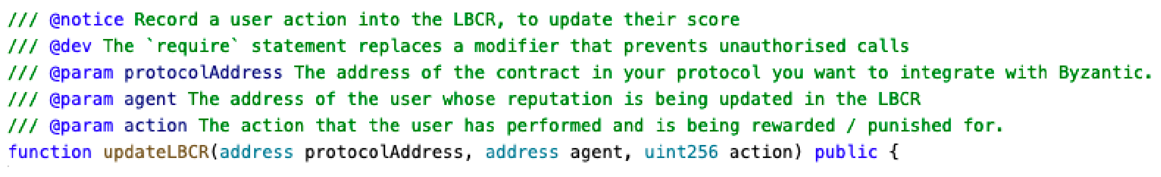
Byzantic Adoption

The aim of this project is to show that reputation can make a significant positive impact in DeFi. We intend to bring this Dapp to market, to reduce collateral and unlock liquidity. For this project gain popularity, it must be as easy as possible for protocols to integrate with Byzantic. To this end, Section 11.1 describes how we automatically generate a documentation using the expressive comments feature of Solidity.

11.1 Automatically generating the documentation

11.1.1 NatSpec

Solidity allows developers to include expressive annotations in the form of comments, that can act as documentation. Such comments are named Ethereum Natural Language Specification Format (NatSpec) [126].
Compiling Solidity smart contracts includes NatSpec annotations in the resulting bytecode, such that certain annotations may be rendered by client software.
It is a recommended coding practice that Solidity smart contracts contain NatSpec that at least documents the publicly available functions. Such comments greatly helped in debugging the double delegatecall, as Figure 5.5 shows.



```
/// @notice Record a user action into the LBCR, to update their score
/// @dev The `require` statement replaces a modifier that prevents unauthorised calls
/// @param protocolAddress The address of the contract in your protocol you want to integrate with Byzantic.
/// @param agent The address of the user whose reputation is being updated in the LBCR
/// @param action The action that the user has performed and is being rewarded / punished for.
function updateLBCR(address protocolAddress, address agent, uint256 action) public {
```

Figure 11.1: NatSpec tags documenting a function in Byzantic

11.1.2 `solidity-docgen`

OpenZeppelin created a tool that automatically generates files based on NatSpec and Handlebars templates, named `solidity-docgen`. Due to the Handlebars, the output format of the documentation is highly customizable. Handlebars compiles templates into functions, and as a result it can generate documentation files from multiple files very quickly.

The command to generate documentation files is:

```
solidity-docgen --solc-module ./node_modules/solc -t docs -s readmes -x adoc
-o docs/modules/ROOT/pages
```

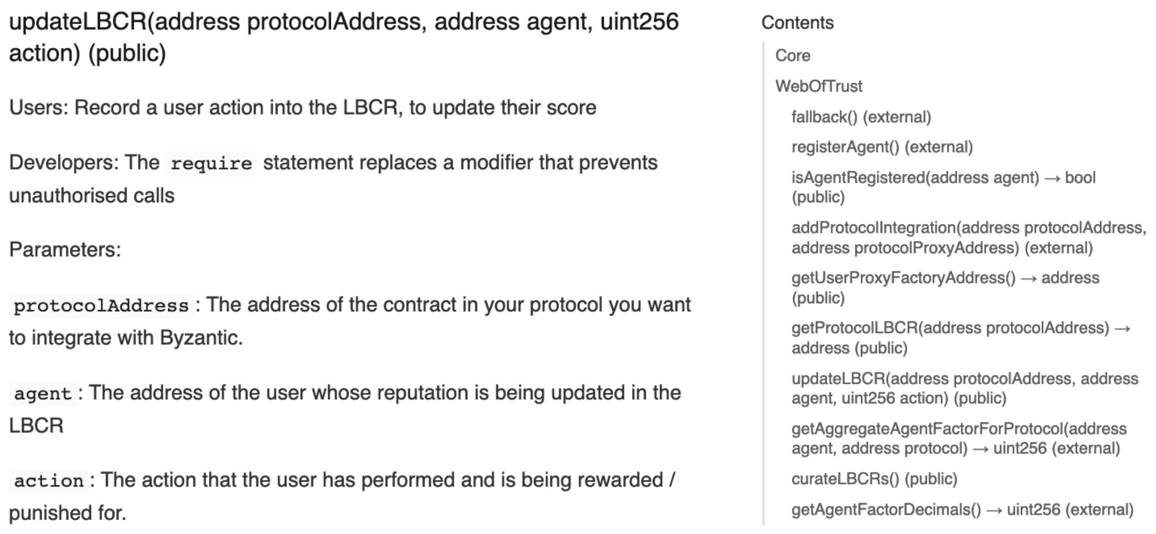
The flags in the command above are:

- `--solc-module`: path to the solc compiler, used convert smart contracts to bytecode and easily extract NatSpec comments
- `-t`: path to directory with Handlebars template files
- `-s`: how contracts should be structured into documentation files. The default value is `contracts`, but it produces hard to decipher documentation files. Instead, we opted for `readmes`, which seems to be what OpenZeppelin are using as well and creates less cluttered files and can be easily parsed by Antora (Section 11.1.3).
- `-x`: extension of documentation files. `adoc` stands for AsciiDoc and is a standard markup language for software documentation. OpenZeppelin use this extension to generate documentation, and we aimed to use the same workflow. AsciiDoc is also particularly easy to convert to a website page.
- `-t`: output directory for documentaiton files.

11.1.3 Antora

Antora [127] is the final step of the documentation generation workflow. It generates static websites from AsciiDoc files, and can combine files coming from multiple repositories. In this case, the tool was run locally, but is otherwise suitable for usage in continuous integration pipelines. The default User Interface template is visually appealing and navigation intuitive (Figure 11.2), but can be changed as needed. After writing an Antora configuration, generating a documentation website is as easy as running:

```
antora antora-playbook.yml
```



updateLBCR(address protocolAddress, address agent, uint256 action) (public)

Users: Record a user action into the LBCR, to update their score

Developers: The `require` statement replaces a modifier that prevents unauthorised calls

Parameters:

`protocolAddress` : The address of the contract in your protocol you want to integrate with Byzantic.

`agent` : The address of the user whose reputation is being updated in the LBCR

`action` : The action that the user has performed and is being rewarded / punished for.

Contents

Core

WebOfTrust

- fallback() (external)
- registerAgent() (external)
- isAgentRegistered(address agent) → bool (public)
- addProtocolIntegration(address protocolAddress, address protocolProxyAddress) (external)
- getUserProxyFactoryAddress() → address (public)
- getProtocolLBCR(address protocolAddress) → address (public)
- updateLBCR(address protocolAddress, address agent, uint256 action) (public)
- getAggregateAgentFactorForProtocol(address agent, address protocol) → uint256 (external)
- curateLBCRs() (public)
- getAgentFactorDecimals() → uint256 (external)

Figure 11.2: Documentation website generated by Antora from the `adoc` file produced by `solidity-docgen`

11.2 Aave Grant Application

We applied to Aave's second round of grants with Byzantic, suggesting that besides unlocking liquidity, our project can be used to improve Aave's new Credit Delegation feature. The new feature works by allowing an agent A to give a part of their balance in Aave to an agent B, for use as collateral. We suggested adding credit scoring to different parties using Byzantic, such that agents who do not know each other can still collaborate based on their reputation. Unfortunately, Aave rejected our application, mentioning that both Byzantic and their Credit Delegation feature were to early on in their development. However, the rejection will not affect our chances of receiving future grants. Aave's response is presented in Figure B.1.

Chapter 12

Conclusion

Byzantic is a privacy-preserving reputation system for DeFi that is applied to reducing over-collateralisation requirements in lending protocols. It is blockchain-agnostic, so the model can be used regardless of where DeFi is implemented. This thesis explored collateral reductions of up to 30% in a protocol with 150% over-collateralisation, Aave (Section 9.4), only requiring users to provide security deposits of 105% of the borrowed value. Lending protocols in DeFi account for 77% of locked-in collateral, which, if reduced by 30%, would create 1.5B USD of additional liquidity as of August 2020. The liquidity created would improve the resiliency of DeFi against future price drops [5].

To the best of our knowledge, this work is the first to propose a configurable reputation system applied to blockchain-wide collateral reduction. It is also the first to use DejaVu, a novel blockchain design pattern for anonymously building reputation (Chapter 4.5).

We have implemented Byzantic in Solidity (Chapter 5) and verified its correctness with smart contract auditing tools (Chapter 10). To encourage adoption, we show examples of how to stress-test different configurations (Chapter 9) and provide an automatically generating documentation (Chapter 11).

12.1 Future Work

Many future opportunities remain for the current project.

- **Under-collateralised loans**. Byzantic currently applies reputation to transforming over-collateralised loans into fully collateralised ones. The next step is going beyond that, by creating a standard Byzantic configuration to be used as a credit scoring system.
- **Real protocol integration**. We plan to get in touch with protocols and integrate them with Byzantic, so that our project actually makes a positive impact in DeFi.

- **Credit delegation**. Aave allows users to delegate their balance so that someone else can use it as collateral [128]. Right now, users need to trust each other informally, but Byzantic reputation could be used to allow strangers to collaborate this way.
- **Dynamic action rewards**. Computing action rewards based on the transacted amount is likely to bring collateral reduction during crises closer to zero.
- **Benefit analysis for less active users**. An analysis of gas costs and user behaviour can be conducted to identify the percentage of users that can benefit from Byzantic beyond the most active 10%.

Appendix A

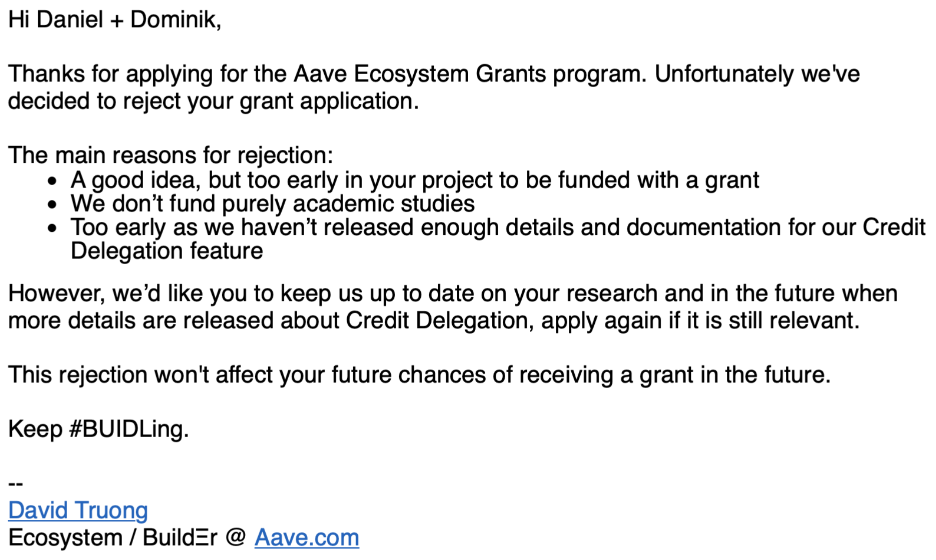
On-chain contracts analysed

Protocol	Address	Description
Compound	0x6c8c6b02e7b2be14d4fa6022dfd6d75921d90e4e	cBAT Token
	0x5d3a536e4d6dbd6114cc1ead35777bab948e3643	cDAI Token
	0x158079ee67fce2f58472a96584a73c7ab9ac95c1	cREP Token
	0xf5dce57282a584d2746faf1593d3121fcac444dc	cSAI Token
	0x39aa39c021dfbae8fac545936693ac917d5e7563	cUSDC Token
	0xf650c3d88d12db855b8bf7d11be6c55a4e07dcc9	cUSDT Token
	0xc11b1268c1a384e55c48c2391d8d480264a3a7f4	cWBTC Token
	0xb3319f5d18bc0d84dd1b4825dcde5d5f7266d407	cZRX Token
	0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5	cETH Token
Aave	0x398ec7346dcd622edc5ae82352f02be94c62d119	LendingPool
Synthetix	0xc011a72400e58ecd99ee497cf89e3775d4bd732f	SNX Token Tracker (now obsolete)

Table A.1: Contracts whose transactions were inspected as part of the behavioural analysis in Chapter 9

Appendix B

Aave Grant Application Response



Hi Daniel + Dominik,

Thanks for applying for the Aave Ecosystem Grants program. Unfortunately we've decided to reject your grant application.

The main reasons for rejection:

- A good idea, but too early in your project to be funded with a grant
- We don't fund purely academic studies
- Too early as we haven't released enough details and documentation for our Credit Delegation feature

However, we'd like you to keep us up to date on your research and in the future when more details are released about Credit Delegation, apply again if it is still relevant.

This rejection won't affect your future chances of receiving a grant in the future.

Keep #BUIDLing.

--
David Truong
Ecosystem / BuildΞr @ Aave.com

Figure B.1: Aave response to our grant application

Bibliography

[1] “The Traditional Financial Institutions”. In: *How to DeFi*. 2020, pp. 4–10.

[2] Timothy C Earle. “Trust, confidence, and the 2008 global financial crisis”. In: *Risk Analysis: An International Journal* 29.6 (2009), pp. 785–792.

[3] Brenda Reddix-Smalls. “Credit Scoring and Trade Secrecy: An Algorithmic Quagmire or How the Lack of Transparency in Complex Financial Models Scuttled the Finance Market”. In: *UC Davis Bus. LJ* 12 (2011), p. 87.

[4] Adrian Zmudzinski. *Joseph Lubin on Ethereum 2.0: ETH to Become 1,000 Times More Scalable Within 24 Months*. May 2019. URL: `https://cointelegraph.com/news/joseph-lubin-on-ethereum-20-eth-to-become-1-000-times-more-scalable-within-24-months`.

[5] Lewis Gudgeon et al. “The Decentralized Financial Crisis: Attacking DeFi”. In: *arXiv preprint arXiv:2002.08099* (2020).

[6] Kalin Nikolov. “A model of borrower reputation as intangible collateral”. In: (2012).

[7] George Lukyanov. *Collateral and Reputation in a Model of Strategic Defaults*. Tech. rep. working paper, 2018.

[8] *Synthetix Litepaper*. URL: `https://www.synthetix.io/uploads/synthetix_litepaper.pdf`.

[9] *DeFi Pulse: The DeFi Leaderboard: Stats, Charts and Guides*. URL: `https://defipulse.com/`.

[10] *The Q1 2020 Ethereum DeFi Report*. URL: `https://consensys.net/blog/news/the-q1-2020-ethereum-defi-report/`.

[11] Dominik Harz et al. “Balance: Dynamic adjustment of cryptocurrency deposits”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1485–1502.

[12] Binance Academy. *The Psychology of Market Cycles*. Jan. 2020. URL: `https://academy.binance.com/economics/the-psychology-of-market-cycles`.

[13] R Houben and A Snyers. *Blockchain: Legal context and implications for financial crime, money laundering and tax evasion. 2018*.

[14] Anand Ranganathan and Roy H Campbell. “What is the complexity of a distributed computing system?” In: *Complexity* 12.6 (2007), pp. 37–45.

[15] *Difference Between a Blockchain and a Database*. Mar. 2020. URL: https://www.ibm.com/blogs/blockchain/2019/01/whats-the-difference-between-a-blockchain-and-a-database/.

[16] *Bitcoin P2P e-cash paper: Satoshi Nakamoto Institute*. URL: https://satoshi.nakamotoinstitute.org/emails/cryptography/1/.

[17] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.

[18] *Blockchain Principles and Applications*. URL: https://www.cs.colostate.edu/~cs481a3/#/.

[19] Arvind Narayanan et al. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[20] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[21] Markus Jakobsson and Ari Juels. "Proofs of work and bread pudding protocols". In: *Secure information networks*. Springer, 1999, pp. 258–272.

[22] Kyle Croman et al. "On scaling decentralized blockchains". In: *International conference on financial cryptography and data security*. Springer. 2016, pp. 106–125.

[23] Rafael Pass, Lior Seeman, and Abhi Shelat. "Analysis of the blockchain protocol in asynchronous networks". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 643–673.

[24] Arthur Gervais et al. "On the security and performance of proof of work blockchains". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.

[25] Mauro Conti et al. "A survey on security and privacy issues of bitcoin". In: *IEEE Communications Surveys & Tutorials* 20.4 (2018), pp. 3416–3452.

[26] Massimo Bartoletti and Roberto Zunino. "BitML: a calculus for Bitcoin smart contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 83–100.

[27] Binance Academy. *Turing Complete - Definition*. May 2019. URL: https://academy.binance.com/glossary/turing-complete.

[28] *What's the Maximum Ethereum Block Size?* Sept. 2019. URL: https://ethgasstation.info/blog/ethereum-block-size/.

[29] *CoinCulture/evm-tools*. URL: https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md.

[30] Daniel Perez and Benjamin Livshits. "Broken Metre: Attacking Resource Metering in EVM". In: *arXiv preprint arXiv:1909.07220* (2019).

[31] *Ethereum*. URL: https://www.sciencedirect.com/topics/computer-science/ethereum.

[32] Petar Maymounkov and David Mazieres. “Kademlia: A peer-to-peer information system based on the xor metric”. In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[33] Seoung Kyun Kim et al. “Measuring ethereum network peers”. In: *Proceedings of the Internet Measurement Conference 2018*. 2018, pp. 91–104.

[34] Yue Gao et al. “Topology Measurement and Analysis on Ethereum P2P Network”. In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2019, pp. 1–7.

[35] Santiago Palladino and OpenZeppelin. *Proxy Patterns*. Mar. 2020. URL: https://blog.openzeppelin.com/proxy-patterns/.

[36] Tornado Cash. *Introducing Private Transactions On Ethereum NOW!* Apr. 2020. URL: https://medium.com/@tornado.cash/introducing-private-transactions-on-ethereum-now-42ee915babe0.

[37] Khaled El Emam and Fida Kamal Dankar. “Protecting privacy using k-anonymity”. In: *Journal of the American Medical Informatics Association* 15.5 (2008), pp. 627–637.

[38] Petar Popovski and Hiroyuki Yomo. “Physical network coding in two-way wireless relay channels”. In: *2007 IEEE international conference on communications*. IEEE. 2007, pp. 707–712.

[39] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O’reilly Media, 2018.

[40] Hannah Murphy. *’DeFi’ movement promises high interest but high risk*. Dec. 2019. URL: https://www.ft.com/content/16db565a-25a1-11ea-9305-4234e74b0ef3.

[41] Lisa Cornish. *Insights from the World Bank’s 2017 Global Findex database*. Apr. 2018. URL: https://www.devex.com/news/insights-from-the-world-bank-s-2017-global-findex-database-92589.

[42] Kyle J Kistner. *How Decentralized is DeFi? A Framework for Classifying Lending Protocols*. May 2020. URL: https://hackernoon.com/how-decentralized-is-defi-a-framework-for-classifying-lending-protocols-90981f2c007f.

[43] *Financial Services Compensation Scheme*. URL: https://www.bankofengland.co.uk/prudential-regulation/authorisations/financial-services-compensation-scheme.

[44] Amitanand S Aiyer et al. “BAR fault tolerance for cooperative services”. In: *Proceedings of the twentieth ACM symposium on Operating systems principles*. 2005, pp. 45–58.

[45] *Collateralized Loans in DeFi*. Nov. 2019. URL: https://defirate.com/collateralized-loan/.

[46] *Saint Fame*. URL: https://www.saintfame.com/.

[47] Colin Harper. *People Are Tokenizing Themselves On Ethereum; Why ”Personal Tokens” Raise Red Flags*. May 2020. URL: `https://www.forbes.com/sites/colinharper/2020/05/06/people-are-tokenizing-themselves-on-ethereum-why-personal-tokens-raise-red-flags/#5526b7306680`.

[48] *Sablier Streams*. URL: `https://docs.sablier.finance/streams`.

[49] Trent McConaghy. *The Layered TCR*. Aug. 2019. URL: `https://blog.oceanprotocol.com/the-layered-tcr-56cc5b4cdc45`.

[50] Ariah Klages-Mundt and Andreea Minca. “While Stability Lasts: A Stochastic Model of Stablecoins”. In: *arXiv preprint arXiv:2004.01304* (2020).

[51] *Compound*. URL: `https://compound.finance/`.

[52] *Aave*. URL: `https://aave.com/`.

[53] *Maker*. URL: `https://makerdao.com/en/`.

[54] *Uniswap V1*. URL: `https://uniswap.org/docs/v1/`.

[55] *Decentralised synthetic assets*. URL: `https://www.synthetix.io/`.

[56] Audun Jøsang, Roslan Ismail, and Colin Boyd. “A survey of trust and reputation systems for online service provision”. In: *Decision support systems* 43.2 (2007), pp. 618–644.

[57] *Secure Collateral Reduction for Many Protocols (trusty)*. Oct. 2019. URL: `https://ethresear.ch/t/secure-collateral-reduction-for-many-protocols-trusty/6377`.

[58] D Harrison McKnight and Norman L Chervany. “The meanings of trust”. In: (1996).

[59] Paul Resnick et al. “Reputation systems”. In: *Communications of the ACM* 43.12 (2000), pp. 45–48.

[60] Eric J Friedman* and Paul Resnick. “The social cost of cheap pseudonyms”. In: *Journal of Economics & Management Strategy* 10.2 (2001), pp. 173–199.

[61] John Kennes and Aaron Schiff. “The value of a reputation system”. In: *Economics working paper archive at WUSTL* (2002).

[62] Bernardo A Huberman and Fang Wu. “The dynamics of reputations”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2004.04 (2004), P04006.

[63] Jordi Sabater and Carles Sierra. “Review on computational trust and reputation models”. In: *Artificial intelligence review* 24.1 (2005), pp. 33–60.

[64] Audun Jøsang and Simon Pope. “Semantic constraints for trust transitivity”. In: *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43*. 2005, pp. 59–68.

[65] Jordi Sabater and Carles Sierra. “REGRET: reputation in gregarious societies”. In: *Proceedings of the fifth international conference on Autonomous agents*. 2001, pp. 194–195.

[66] Jordi Sabater. “Evaluating the ReGreT system”. In: *Applied Artificial Intelligence* 18.9-10 (2004), pp. 797–813.

[67] Jiangshan Yu et al. “Repucoin: Your reputation is your power”. In: *IEEE Transactions on Computers* 68.8 (2019), pp. 1225–1237.

[68] Matt Blaze, Joan Feigenbaum, and Jack Lacy. “Decentralized trust management”. In: *Proceedings 1996 IEEE Symposium on Security and Privacy*. IEEE. 1996, pp. 164–173.

[69] Taher Elgamal and Kipp EB Hickman. *Secure socket layer application program apparatus and method*. US Patent 5,657,390. Aug. 1997.

[70] Tyrone Grandison and Morris Sloman. “A survey of trust in internet applications”. In: *IEEE Communications Surveys & Tutorials* 3.4 (2000), pp. 2–16.

[71] *DeFi Pulse: The DeFi Leaderboard: Stats, Charts and Guides*. URL: `https://defipulse.com/`.

[72] *What Is Liquidity And How Does It Affect Prediction Markets?: Cultivate Labs Blog*. URL: `https://www.cultivatelabs.com/posts/what-is-liquidity-and-how-does-it-affect-prediction-markets`.

[73] Miles Brundage et al. “The malicious use of artificial intelligence: Forecasting, prevention, and mitigation”. In: *arXiv preprint arXiv:1802.07228* (2018).

[74] Chris Burnett, Timothy J Norman, and Katia Sycara. “Bootstrapping trust evaluations through stereotypes”. In: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*. International Foundation for Autonomous Agents and Multiagent Systems. 2010.

[75] Xin Liu et al. “Stereotrust: a group based personalized trust model”. In: *Proceedings of the 18th ACM conference on Information and knowledge management*. 2009, pp. 7–16.

[76] Taha Gunes, Long Tran-Thanh, Timothy Norman, et al. “Identifying vulnerabilities in trust and reputation systems”. In: (2019).

[77] Don Ross. *Game Theory*. Mar. 2019. URL: `https://plato.stanford.edu/entries/game-theory/`.

[78] Ittay Eyal and Emin Gün Sirer. “Majority is not enough: Bitcoin mining is vulnerable”. In: *International conference on financial cryptography and data security*. Springer. 2014, pp. 436–454.

[79] Philip Daian et al. “Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges”. In: *arXiv preprint* (2019).

[80] ConsenSys. *Known Attacks*. URL: `https://consensys.github.io/smart-contract-best-practices/known_attacks/#forcibly-sending-ether-to-a-contract`.

[81] William Foxley. *Everything You Ever Wanted to Know About the DeFi ’Flash Loan’ Attack*. Feb. 2020. URL: `https://www.coindesk.com/everything-you-ever-wanted-to-know-about-the-defi-flash-loan-attack`.

[82] *Solidity Documentation*. URL: `https://solidity.readthedocs.io/en/v0.6.8/`.

[83] Weiqin Zou et al. "Smart contract development: Challenges and opportunities". In: *IEEE Transactions on Software Engineering* (2019).

[84] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

[85] *The GNU Debugger*. URL: `https://www.gnu.org/software/gdb/`.

[86] *Buidler*. URL: `https://buidler.dev/`.

[87] *What is CI/CD?* URL: `https://www.redhat.com/en/topics/devops/what-is-ci-cd`.

[88] *The Lean Startup Methodology*. URL: `http://theleanstartup.com/principles`.

[89] Benedikt Bünz et al. "Zether: Towards privacy in a smart contract world". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2020, pp. 423–443.

[90] Zachary J Williamson. *The aztec protocol*. 2018. URL: `https://github.com/AztecProtocol/AZTEC`.

[91] *ZK Proof Community Reference*. URL: `https://docs.zkproof.org/reference#latest-version`.

[92] *Ethereum for Developers*. URL: `https://ethereum.org/en/developers/`.

[93] *New York Blockchain Week*. URL: `https://gitcoin.co/hackathon/new-york-blockchain-week/?org=aave&tab=hackathon:20`.

[94] Truffle Suite. *Ganache*. URL: `https://www.trufflesuite.com/ganache`.

[95] *Ethereum API: IPFS API Gateway: ETH Nodes as a Service*. URL: `https://infura.io/`.

[96] *Aave LendingPoolAddressesProvider*. URL: `https://docs.aave.com/developers/developing-on-aave/the-protocol#lendingpooladdressesprovider`.

[97] Vitalik Buterin. *EIP 7: DELEGATECALL*. Nov. 2015. URL: `https://eips.ethereum.org/EIPS/eip-7`.

[98] *djrtwo/evm-opcode-gas-costs*. URL: `https://github.com/djrtwo/evm-opcode-gas-costs/blob/master/opcode-gas-costs_EIP-150_revision-1e18248_2017-04-12.csv`.

[99] *ETH Gas Station*. URL: `https://ethgasstation.info/`.

[100] *Gwei to USD Conversion*. URL: `https://www.cryps.info/en/Gwei_to_USD/`.

[101] Trufflesuite. *Truffle debug with external contracts · Issue 2970 · trufflesuite/truffle*. URL: `https://github.com/trufflesuite/truffle/issues/2970`.

[102] Trufflesuite. *Enhancement: Allow debugger to download and debug external sources off Etherscan by haltman-at · Pull Request 3085 · trufflesuite/truffle*. URL: `https://github.com/trufflesuite/truffle/pull/3085`.

[103] Sean Bowe, Ariel Gabizon, and Ian Miers. "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model." In: *IACR Cryptol. ePrint Arch.* 2017 (2017), p. 1050.

[104] Yuval IshaiAll author posts et al. *Zero-Knowledge Proofs from Information-Theoretic Proof Systems - Part I*. Aug. 2020. URL: `https://zkproof.org/2020/08/12/information-theoretic-proof-systems/`.

[105] *SUMMARY OVERVIEW OF STABLECOINS AND THE LAW REGARDING STABLECOINS*. 2019. URL: `https://www.cftc.gov/media/2731/TAC100319_Stablecoins/download`.

[106] *Uniswap Whitepaper*. URL: `https://uniswap.org/whitepaper.pdf`.

[107] Will Heasman. *Are the BZx Flash Loan Attacks Signaling the End of DeFi?* Feb. 2020. URL: `https://cointelegraph.com/news/are-the-bzx-flash-loan-attacks-signaling-the-end-of-defi`.

[108] Garth Travers. *Proxy contract cutover on May 10*. May 2020. URL: `https://blog.synthetix.io/proxy-contract-cutover-on-may-10/`.

[109] Andrew Miller et al. "An empirical analysis of linkability in the monero blockchain". In: *arXiv preprint arXiv:1704.04299* (2017).

[110] Jiawei Han, Mieheline Kamber, and J Pei. *Data mining techniques and concepts*. 2006.

[111] Nathan Harness and Lloyd Alty Luke Goldsmith. *The psychology of stock market cycles*. URL: `https://www.delawarefunds.com/insights/the-psychology-of-stock-market-cycles`.

[112] *How to Deal with Cryptocurrency FOMO*. URL: `https://vocal.media/theChain/how-to-deal-with-cryptocurrency-fomo`.

[113] Terje Aven. "On the meaning of a black swan in a risk context". In: *Safety science* 57 (2013), pp. 44–51.

[114] *Why Dai is the Most Used Cryptocurrency in the DeFi Space*. Aug. 2020. URL: `https://blog.makerdao.com/why-dai-is-the-most-used-cryptocurrency-in-the-defi-space/`.

[115] Alejandro MiguelAlejandro is a New-Zealand based trader, writer who has been involved in the cryptocurrency, and blockchain space since early 2016. Being extremely passionate about this emerging technology. *ETH Lending FAQ*. Mar. 2020. URL: `https://defirate.com/eth/`.

[116] Sc-Forks. *sc-forks/solidity-coverage*. URL: `https://github.com/sc-forks/solidity-coverage`.

[117] *Test Coverage History and Statistics*. URL: `https://coveralls.io/github/OpenZeppelin`.

[118] Asem Ghaleb and Karthik Pattabiraman. "How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection". In: *arXiv preprint arXiv:2005.11613* (2020).

[119] Loi Luu et al. “Making smart contracts smarter”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 254–269.

[120] Petar Tsankov et al. “Securify: Practical security analysis of smart contracts”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 67–82.

[121] ConsenSys. *ConsenSys/mythril*. URL: `https://github.com/ConsenSys/mythril`.

[122] Sergei Tikhomirov et al. “Smartcheck: Static analysis of ethereum smart contracts”. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2018, pp. 9–16.

[123] Mark Mossberg et al. “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1186–1189.

[124] Josselin Feist, Gustavo Grieco, and Alex Groce. “Slither: a static analysis framework for smart contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE. 2019, pp. 8–15.

[125] Jim Mueller. *Learn about Financial Liquidity*. Jan. 2020. URL: `https://www.investopedia.com/articles/basics/07/liquidity.asp`.

[126] *NatSpec Format*. URL: `https://solidity.readthedocs.io/en/v0.5.10/natspec-format.html`.

[127] OpenDevise Inc. URL: `https://antora.org/`.

[128] *Credit Delegation*. URL: `https://docs.aave.com/developers/developing-on-aave/the-protocol/credit-delegation`.