Dissimilar Redundancy in DeFi

Daniel Perez
Imperial College London

Lewis Gudgeon
Imperial College London

***Abstract*—One of the most salient features of the meteoric rise of Decentralized Finance (DeFi), has been frequent and often financially devastating protocol attacks. As of [May 2022], there have been over 70 exploits of DeFi protocols, with the total of lost funds amounting to approximately 1.5bn USD. In this paper, we introduce a new approach to minimizing the frequency and severity of such attacks: dissimilar redundancy for smart contracts. In a nutshell, the idea is to implement a program logic *more than once*, ideally using *different* programming languages. Then, for each implementation, the results should *match* before allowing the state of the blockchain to change. This is inspired by and has clear parallels to the field of avionics, where on account of the safety-critical environment, flight control systems typically feature multiple redundant implementations. We argue that the high financial stakes in DeFi protocols merit a conceptually similar approach, and we provide a novel algorithm for implementing dissimilar redundancy for smart contracts.**

I. Introduction

Decentralized Finance (DeFi) Protocol hacks are frequent - with more than 70 to date totalling losses of more than 1.5bn USD [1]. For a financial infrastructure that is purportedly going to replace traditional finance, this is worrisome. The severity of the issue of hacks is exacerbated by the non-custodial nature of DeFi systems. Unlike in traditional financial systems, where in the event of financial disaster, there are often safety-nets such as the state or insurers, in the DeFi setting there are no such safety provisions at scale. Moreover, while there is a nascent insurance market for DeFi insurance, e.g. [2], such solutions provide only a (necessary) second-best solution: providing coverage in the event of a DeFi system failure. A first-best solution is to prevent the failure in the first place.

First-best prevention is, however, challenging. Leaving aside the security of the underlying blockchain layer, the two main pillars of DeFi protocol security are (i) extensive smart contract testing and (ii) code audits. Both of these have drawbacks. Ensuring adequate test coverage is very challenging, not least when the objective is to ensure all edge-cases are covered in the tests. While testing is an essential part of smart contract development, we suggest that it is not realistic to expect even best practice testing with a very high degree of coverage to be sufficient to prevent *all* bugs. Code audits are often performed by external teams under time pressure, with audits typically lasting 2-3 weeks from start to finish, even for teams with no prior familiarity with the code base. Even for the most experienced software engineer, native to DeFi, it is wishful thinking to believe that they will always be able to catch every bug. The problem is compounded by smart contract composability - where DeFi protocols are snapped together like DeFi lego - which serves to increase the the challenge as tests and auditors now have to anticipate bugs that could arise with as yet unseen smart contracts.

This paper presents a new approach to preventing failures at the smart contract level: dissimilar redundancy. In aviation, the safety-critical nature has led to the emergence of a practice of implementing multiple separate and redundant flight control systems. For example, the Boeing 777 [3] featured a Fly-By-Wire flight control system that had to meet extremely high levels of functional integrity and reliability. To do this, it had three separate primary flight computers, with each computer containing three *dissimilar* internal computational lanes. The lanes differed in terms of compilers, power supply units and microprocessors, with, for example, lane 1 using the AMD 29050, lane 2 the Motorola 68040 and lane 3 the Intel 80486. Within each of the three flight computers, two of the lanes acted as monitors while the third lane was in command. In this way, the flight computer features a form of redundancy that is *dissimilar*, with the multiple lanes being resistant to bugs induced by microprocessors or compilers.

We apply the core of this idea to smart contracts. We implement and detail a system based on a proxy pattern which relies on dissimilar implementations of a DeFi protocol, and cross-checks one against the other before effecting any on-chain state change. On Ethereum, this approach has already been taken for client implementations, with the community maintaining multiple open-source clients, developed by different teams and using different programming languages [4]. The purpose of this approach is to strengthen the network and make it more diverse, with a view to avoiding a single client dominating the network in order to remove single points of failure. We extend this concept to the smart contract layer itself.

Our contributions are as follows.

- We introduce the notion of dissimilar redundancy for DeFi protocols
- We provide the first implementation of a protocol for dissimilar redundancy for a DeFi protocol[1]
- We evaluate the protocol on a smart contract auction system implemented in both Solidity and Vyper, verify that a fuzzing approach would be able to detect purposefully introduced bugs, and provide the costs in USD of using a protocol for dissimilar redundancy

[1] https://github.com/danhper/smart-contract-dissimilar-redundancy

Outline

We set out the necessary background in Section II, provide our methodology in Section III, evaluate it in Section IV, consider related work in Section VI and conclude in Section VII.

II. Background

In this section, we provide the necessary background regarding smart contracts, their potential vulnerabilities, as well as the cost of their execution.

A. Smart contracts

Smart contracts are program objects that are native to blockchains. In the context of Ethereum, such smart contracts require that the underlying blockchain is a transaction-based state-machine. State changes occur on a blockchain through transactions, which are atomic: they either succeed, where the state is updated, or fail, where the state of the blockchain remains unchanged. For the approach we take below, atomicity is a crucial property of smart contracts as it provides that the blockchain cannot be left in an invalid or inconsistent state. Communication between two smart contracts occurs via message calls within the same execution context.

B. Scaling solutions

Given the high cost of using Ethereum, many scaling solutions have emerged trying to solve this issue. At the heart of these approaches is taking computation off the layer-one blockchain, and instead performing this on a separate later while using the layer-one blockchain as an anchor of trust. Such protocols are typically denoted layer two protocols, the name for a family of solutions that seek to allow applications to scale by processing transactions off the main blockchain.

In the context of Ethereum, one approach is to use rollups [5]. Rollups perform transaction execution away from the layer-one blockchain but store transaction data on-chain. In doing so, the security properties of layer-one are leveraged as an anchor of trust by the rollup approach, which is then able to perform transaction execution off-chain.

For a full summary of these approaches, see [6].

C. Smart contract vulnerabilities

Over the years, there have been many smart contract exploits often leading to significant losses of money [7], [8]. There are many different ways in which smart contracts can be exploited but on a very high level, attacks can be classified as "technical", which means that an attacker can steal funds by exploiting a technical issue, such as a bug, in the contract, or "economical", which means that the attacker can exploit a contract through incentive that might not be properly aligned [9]. For the purpose of this paper, we will focus on technical security of smart contracts. Within technical security itself, there is also a myriad of different potential issues, such as contract vulnerable to re-entrancy [10] or flash loan attacks [11]. However, another very common way in which contracts fail is simply logical bugs. Such logical bugs have cost millions to many smart contract projects. For example, Compound was recently victim of such an incident [12], that cost the protocol over 90 million USD. The error was as simple as a greater than symbol that should have been greater or equal.

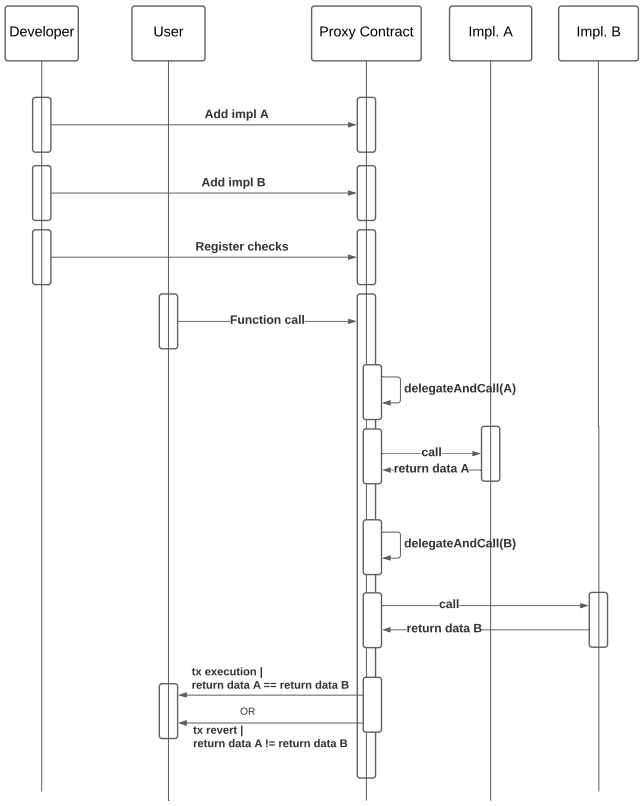

Fig. 1: Overview of our dissimilar redundancy framework

III. Methodology

A. Overview

We now turn to how we use the approach of dissimilar redundancy in the context of Ethereum. At the centre of our approach is the use of a proxy architecture pattern. With a proxy pattern, all message calls to a contract $\mathcal{C}$ first go through a proxy contract $\mathcal{P}$ that serves to direct the message calls to contract $\mathcal{C}$. With this pattern, contract $\mathcal{C}$ contains the actual implementation logic while contract $\mathcal{P}$ provides a storage layer. At present, a common use of this pattern is to provide contract upgradeability [13], [14]: while contracts cannot be directly upgraded once deployed, upgradeability can be mimicked by changing where contract $\mathcal{P}$ delegates calls to from contract $\mathcal{C}$ to contract $\mathcal{C}_1$.

We expand on this pattern: in our pursuit of dissimilar redundancy, we allow $\mathcal{P}$ to delegate to multiple implementations at once. This is in contrast to the standard pattern which only permits delegation to a single implementation at a time. In a nutshell, our proxy contract $\mathcal{P}$ sequentially calls two different implementations - supposedly identical in logic - $\mathcal{C}_1$ and $\mathcal{C}_2$ and ensures that the data returned by function calls to each implementation as well as return values from an arbitrary

Algorithm 1 Dissimilar redundancy framework call delegation

```
function CallDelegate(impl, data, checks, isLast)
    (ok, retData) ← DelegateTo(impl, data)
    checkResults ← RunChecks(checks)
    checksHash ← HashChecks(checkResults)
    if isLast then
        return (ok, retData, checksHash)
    else
        revert (ok, retData, checksHash)
    end if
end function

function RedundantCall(implementations, data)
    n ← Length(implementations)
    signature ← GetSig(data)
    checks ← GetChecks(signature)

    for i ← 0, n − 1 do
        impl ← implementations[i]
        last ← i == n − 1

        callData ← Encode(CallDelegate, impl, data, checks, last)
        (_, delegateRet) ← DelegateTo(this, callData)
        (ok, retData, checksHash) ← Decode(delegateRet)

        if IsDefined(previousOk) then
            Assert(previousOk == ok)
            Assert(previousRetData == retData)
            Assert(previousChecksHash == checksHash)
        else
            previousOk ← ok
            previousRetData ← retData
            previousChecksHash ← checksHash
        end if
    end for

    return (ok, retData)
end function
```

number of *checks* provided by the contract developer match. In this context, a check is a call to a contract's function of which the return value is deemed relevant to the function called. For example, in the context of an ERC-20 token [15], the developer might want to add `balanceOf(from)` and `balanceOf(to)` as a check for `transferFrom(from, to, amount)`. The proxy will then call `balanceOf` twice after each call to `transferFrom` and ensure that the results are consistent among the implementations.

Although the overall idea is straightforward, the actual implementation requires several technical difficulties to be overcome. We show an overview of the framework in Figure 1.

B. The technical challenges

a) Calling implementations with the same state: The first difficulty is that our approach requires both implementations to be called with the same initial state, sequential contract calls would typically modify this state. To overcome this, all state changes made by a call to an implementation need to be rolled back before we call the next implementation. We leverage the atomic nature of Ethereum's calls to achieve this: if a transaction raises an error, the state reverts to the initial state.

In our approach, instead of the proxy directly delegating to an implementation, it first delegates to *itself*, passing in the call data as an argument along with the implementation to call and the checks to perform. In this delegated call, the proxy then delegates to the implementation, executes all the checks and combines their result in a single hash value. It then reverts the execution to rollback the changes made by the implementation and returns the checks hash as the revert data. The only exception to this is the call to the last implementation, where it returns the checks hash normally instead of reverting, to persist the changes. We provide a high-level overview of the delegation logic in Algorithm 1. Low-level implementation details can be found in our open-source implementation.

b) Check encoding: The second difficulty concerns how the checks to be performed after each execution should be encoded. A check is a call to a contract that needs to be consistent after each execution. To make the proxy retain the interface of a proxied implementation, we must pre-register the checks with the proxy, rather than specifying the checks with each call. A naive implementation would permit the developer to register a function with pre-encoded arguments to be called after any call. However, such an implementation would have severe limitations. Pre-encoding arguments makes calls such as the one described above for `balanceOf` *impossible*, as these depend on call data and transaction information—this information is only available at runtime. Since these calls need to be well targeted for them to be effective and find potential discrepancies among implementations, such an approach would not be viable.

Instead, we implement an approach which achieves the following objectives:

1) the registration of per-function checks, without pre-encoded arguments. For example: `transferFrom` and `approve` could have a different set of checks registered.
2) call data and transaction information should be accessible during the checks

The first objective is easily achieved by storing a mapping from function signature to checks and retrieving the relevant checks depending on the function signature called by the current call to the proxy. The second objective is more challenging, as the developer must be able to register checks upfront that rely on information only available when the function is executed. To allow for this, rather than registering checks by passing in the arguments themselves, we design a simple byte encoding for the arguments that allows abstract arguments, registered with the checks, to be mapped to the concrete arguments that are computed when executing the checks. For example, an abstract argument could be "the first argument of the current call" or "the sender of the current transaction". When executing the checks, the proxy will map these to the actual value of the first argument or the sender of the transaction, and encode these when calling the check function.

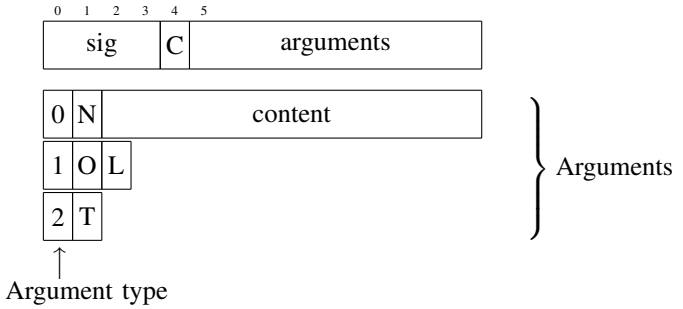

Fig. 2: Encoding for registering checks. C is the number of arguments. The three types of arguments are shown. Type 0 represents a static argument and N is the length of the static content. Type 1 represents call data argument where O and L are the offset and length to retrieve from call data. Type 2 represents environment argument and T is the type of environment variable (e.g. `msg.sender`) to use

```
tokenProxy.registerCheck(sig["transferFrom"],
  tokenProxy, encode_args(Token, "balanceOf", [
  (ArgumentType.CallData, (4, 32))]))

tokenProxy.registerCheck(sig["transferFrom"],
  tokenProxy, encode_args(Token, "balanceOf", [
  (ArgumentType.CallData, (36, 32))]))

tokenProxy.registerCheck(sig["transferFrom"],
  tokenProxy, encode_args(Token, "allowance", [
  (ArgumentType.CallData, (4, 32)),
  (ArgumentType.Env, EnvArg.Sender)]))
```

Fig. 3: Example checks registration for an ERC-20 token `transferFrom` function

In Figure 2, we show how we encode abstract arguments to register the checks. The first four bytes are, as for regular calls, the signature of the function to be called. The next byte is the number of arguments to pass to the function. Each argument can then be one of three types: a static argument, a call data argument or an environment argument. These types determine how the concrete argument will be computed when the contract is called:

- A static argument: simply passed through to the function as a concrete argument
- A call data argument: the given bytes from the transaction call data are extracted and passed as an argument
- An environment argument: looks up the concrete argument in the current transaction. The argument to be looked up in the environment, e.g., the sender of the transaction or the current block timestamp is specified by a single byte in the abstract argument.

Once all the abstract arguments are converted into their concrete counterparty, they are encoded along with the function signature and the check can be performed.

This encoding provides enough flexibility to perform a wide variety of different checks.

c) Toy example: We show an example of such checks in Figure 3, where we register three different checks for the `transferFrom` function of an ERC-20 token. The two first checks are for the balance of the first argument (the address sending tokens) and the second argument (the address receiving tokens) of the `transferFrom`. In both cases, we extract these arguments from the call data. The third check is for the allowance and also uses the first argument but also the sender of the transaction, as their allowance is expected to decrease after a successful call to `transferFrom`.

Now that we have established the methodology behind this approach to implementing dissimilar redundancy, we turn to an evaluation.

IV. Evaluation

To evaluate our solution, we use a smart contract implementing a simple auction system. The rules of the auction system are as follow:

1) A seller starts an auction with an NFT of their choice and sets an end time
2) The NFT is transferred to the auction contract
3) Any user can bid in the auction and the bid must be strictly greater than the previous highest until the auction ends
4) After the auction ends, anyone can "finalize" the auction, which will either transfer the NFT to the winner of the auction or transfer it back to the owner if there were no bids

We implement the auction contract with the same behavior in both Solidity and Vyper but purposefully introduce a couple of implementation bugs in one of the two contracts. We then check whether these bugs would be detected by fuzzing the contract and causing the proxy to fail due to inconsistencies in the evaluation results.

In particular, we introduce two bugs to the Vyper version of the contract. First, rather than checking than the bid is strictly greater than the previous one, we check that the bid is greater or equal. This means that in this particular case, the Solidity version will revert the transaction while the Vyper one will successfully execute. Second, we omit the case where there were no bidders. As a result, both versions will successfully execute but for the Vyper implementation, the ownership of the NFT will still be the auction contract in the case there were no bidders.

A. Development-time testing

We first test our code locally to show how our approach can make bug-detection at development time significantly easier.

To test our code, we use Python in combination with the Brownie framework [2] for testing and the Hypothesis library [16] to generate test cases. We show the most important part of the code we use to our auction contract in Figure 4. We note that the `ensure_consistent` context manager is

[2]https://eth-brownie.readthedocs.io/

```
auction_proxy.registerCheck(sig["finalize"],
  nft_collection.address,
  encode_args(
    NftCollection, "ownerOf",[
    (ArgumentType.Static, ("uint256", NFT_ID))]
))

@given(bids=st.lists(st.tuples(
   st.integers(min_value=0), addresses())))
def test_bid(bids):
  with ensure_consistent():
    for value, account in bids:
        auction_proxy.bid({"from": account,
                           "value": value})

@given(bids=st.lists(st.tuples(
   st.integers(min_value=0), addresses())))
def test_finalize(bids):
  with ensure_consistent():
    for value, account in bids:
      auction_proxy.bid({"from": account,
                         "value": value})
    chain.sleep(3600)
    auction_proxy.finalize()
```

Fig. 4: Testing code using dissimilar redundancy

```
Falsifying example: test_bid_consistency(
    bids=[
     (1, <Account '0x66aB6...5871'>),
     (1, <Account '0x66aB6...5871'>)],
)
=============================================
FAILED tests/test_auction.py::test_bid -
brownie.exceptions.VirtualMachineError: revert:
all implementations must return the same success
```

(a) Failing test case for the `bid` function showing a failure after two bids with the same value were placed

```
Falsifying example: test_finalize_consistency(
  bids=[]
)
=============================================
FAILED tests/test_auction.py::test_finalize -
brownie.exceptions.VirtualMachineError: revert:
all implementations must return the same checks
```

(b) Failing test case for the `finalize` function showing a failure when no bids have been placed

Fig. 5: Failing tests when fuzzing the auction contract with a correct and an incorrect implementation

implemented as part of our tooling and will only fail if a call reverts because implementations did not behave similarly.

In the tests, we first register a check for finalize that will look up the owner of the NFT that was on sale. For testing both `bid` and `finalize`, we generate random bids, where a bid is an account and value (price to pay for the auction) pair. For bid, we only execute all the bids while for finalize, we also ensure that the auction is ended and execute finalize.

Using this approach, trying to fuzz the contract requires performing differential fuzzing on the different implementations of the contract logic registered by the proxy. This makes it possible to easily identify the cases where the two implementations do not behave in the same way.

In Figure 5, we show the results of these tests. Note that we fix the first failing test before proceeding to the second one.

For the `bid` function, the test framework correctly outputs a failure when two bids are placed with the same value in a row. The failure scenario is clear from the test output, as the two bids of the input both have a value of 1. The error message mentions that all implementations should return the same "success", which means that one of the implementations successfully executed (the bug-ridden version), while the other failed to executed.

The test for the `finalize` function also correctly fails with an example containing no bids. This is consistent with the bug in the Vyper version of the Solidity which does not transfer back the token properly to its original owner. The failing test also mentions that all implementations must return the same checks which means that all implementations had the same success status (they all succeeded in this particular case) but the checks did not return the same value. Indeed, a check was registered to look up the NFT owner.

Overall, with this example, we have seen that with only a few lines of code, it was possible to have extensive coverage of the tested function that is able to automatically find discrepancies among implementations and indicate to the developer the test cases that would yield different results.

Name	Address
Auction proxy	`0xAd837BDD116C14aA82311Db7D1879C7cDDCfd283`
Auction (Sol, proxied)	`0xdb85f3DB2aA6E5e294485972ABE921be188b6A37`
Auction (Vy, proxied)	`0x9FD31161360B5E772f2b9C469D4A35E679273Dbf`
Auction (Sol, standalone)	`0xE575CCb0213393eBFc9258013af1c43e9E416544`
Auction (Vy, standalone)	`0xEe164319fE07127Efc8fdf8b3e99ea736F8c955E`

TABLE I: Contracts deployed on Polygon mainnet. "Sol" are Solidity contracts. "Vy" are Vyper contracts. "Proxied" are contracts used by the proxy. "standalone" are contracts interacted with directly.

B. Real-world deployment

An important strength of our approach is that it is possible to utilize the two implementations not only at development and testing time but also after the contract is deployed, ensuring that all the transactions executed will always be consistent across the different implementations provided. To demonstrate how this would perform on a real-world blockchain, we deploy our auction and its two implementations on the Polygon main network, ensure that the calls that would trigger revert correctly, and measure the cost overhead of our approach. We provide a list of all the deployed contracts in Table I.

To be able to give comparable results, we deploy our proxy using the Solidity and Vyper implementations, as well as a standalone version of each implementation. During our interactions with the contracts, we maintain a fixed gas price

	Start	First bid	Subsequent bid	Finalize
Proxied	0.0229	0.0092	0.006	0.0144
Solidity	0.0094	0.0045	0.0029	0.0052
Vyper	0.0108	0.0045	0.0029	0.0071

TABLE II: USD cost[3] of calling different functions of the auction contract with and without dissimilar redundancy proxy. The gas price is fixed to 30 Gwei.

of 30 Gwei, which at the time of writing was enough for near instantaneous inclusion in a Polygon block.

We summarize the cost nominated in US dollars of all the interactions with our auction contracts in Table II. We split the costs of the first and the subsequent bids, since the first bid allocates storage, using more gas than subsequent ones. Since the proxy is using both the Solidity and the Vyper implementation, a lower bound for the cost is the sum of the cost of each individual implementation. The difference between the sum of these costs and the cost of calling the proxy is the overhead of the proxy itself, including the cost of calling checks after each call and checking consistency among results. In our example, only `finalize` has a check registered, to check for the owner of the NFT after execution. For the calls to `start` and `bid`, respectively about 1.2% and 2.2% of the total cost is part of the proxy overhead, the rest of the cost being used by the actual underlying functions. For `finalize`, the overhead is understandably higher as a check is registered. Indeed, about 14.5% of the cost is used by the proxy itself.

We also check that a transaction that would cause our correct and our buggy implementation to diverge would correctly revert. To do so, we bid using subsequently twice the same amount, which fails for the proxied version (tx `0x5f840a...6dc334c`) by returning the same error as the one we saw during the tests. As a sanity check, we try the same sequence of bids on the standalone implementations and the Solidity version reverts correctly (tx `0x406ad7...7759673`) while the Vyper one succeeds (tx `0x888189...7df41e8`).

V. Limitations

While our approach comes with strong benefits in terms of reliability, it has some limitations. In this section, we will go over these limitations and provide details on how some of them could be tackled.

Transaction fees. As we have seen in the previous section, this approach will always at least double the cost of transactions and potentially increase it further if many calls need to be performed. There is not any direct way to prevent this issue or improve it significantly at the implementation level. This means that using such an approach on a expensive network, such as Ethereum, is almost unfeasible, as the price increase for transactions would likely be unacceptable for many users. However, more and more layer two solutions are being developed, with transaction fees orders of magnitude lower than what can be seen on Ethereum mainnet. This is for example the case of Polygon mainnet, which we used for the evaluation earlier. As we can see in Table II, although the costs have been doubled, this represents an increase in the order of a cent in the worst case. As Ethereum is moving towards a layer 2 future [17] and transaction fees become negligible, the fee overhead of our approach might become less and less of a concern.

Development cost. Another obvious limitation is the development cost of another implementation of the same contract. However, there are several arguments why this might not be a critical issue.

First, such an approach has been seen with Ethereum clients, which are vastly larger in terms of complexity and code base than most protocols built on top of smart contracts. In a similar way that the Ethereum Foundation distributes fund to help external teams maintain clients, a well established protocol could potentially operate similarly to have alternative implementation maintained.

Second, given the often enormous amount of money at stake, a significant part of the smart contract development goes into testing rather than simply implementing the contract. For example, Uniswap V3 [18] has almost twice more test code than actual implementation. Our approach being highly complementary to regular testing, it could be integrated as part of the development process as yet another way to reduce the number of potential bugs or vulnerabilities in the contract.

Storage layout. Another limitation of our approach is that all the implementations must use exactly the same storage layout. The proxy contract will be storing all the state of the contract, so all implementations must read at the same location in storage to be able to retrieve the information they are looking for. One of the main drawback is that it imposes some rigidity on alternative implementations, which might make it less likely to try implementing the logic in very similar way and potentially replicate bugs in multiple implementations. Another issue is that this makes it hard to combine implementations written in Solidity and Vyper. The two languages use mostly the same mapping from state variable to Ethereum storage slot but for mappings, they use a very slightly different way to compute the storage slot which makes it impossible to use this approach for any contract using mappings. However, this issue is trivial to fix, as it would only require a very minor, albeit backward incompatible, change in the Vyper compiler.

VI. Related work

We first present how a similar approach has been helpful for Ethereum clients and then present some related research discussing differential fuzzing.

Ethereum clients. From early on, the Ethereum network has been running using different client implementations. The goal of having multiple clients has always been to make the network more robust and ensure that it does not fail in case there would be a bug, a vulnerability, or an avenue for a potential denial-of-service (DoS) attack in one of the implementations. In the case

[3]We use the December 12th 2021 price of 2.15 USD per MATIC token, Polygon's native token used to pay for gas fees

of Ethereum, this allows for several failure modes. If one of the implementations had a bug that would make it crash on certain transactions, the network would continue to operate with other implementations that do not contain this bug. This would be similar in the case of a DoS attack only effective on one type of implementation. On the other hand, if a bug or exploit would result in a different state transition after executing a transaction, it would create a network split where the victim implementation would only manage to reach consensus with nodes using the same implementation. This is riskier than the previous case but remains safe as long as exchanges and other entities bridging on-chain activity to off-chain assets ensure all implementations are in a consistent state before accepting to process a transaction. Overall, this diversity of clients has been very beneficial to Ethereum, despite the high maintenance cost, and has allowed it to operate smoothly for over 6 years.

Differential fuzzing. Having two implementations that should behave in the same way allows to perform differential fuzzing: fuzzing both implementations trying to look for cases where they would behave differently. This technique has already been used in multiple domains such as cryptography [19], programming languages [20], and blockchain consensus [21], [22]. A recent work leveraging differential fuzzing to find bug in Ethereum clients [23] has managed to find not only most known consensus bugs but also two new ones, including a bug that led to a fork in the consensus due to only part of the full nodes in the network having upgraded to the latest version [24]. Overall, this shows the potential of differential fuzzing and how it can be useful for finding bugs and zero-day exploits.

VII. Conclusion

We have argued that the high financial stakes in the context of DeFi merit an approach to program redundancy inspired by avionics: the utilization of dissimilar redundancy. Through implementing the same program logic more than once, ideally with different programming languages and even by different engineering teams, and then using an on-chain execution logic that ensures that the dissimilar implementations must *agree* before the on-chain state can update, redundancy is brought into the smart contract ecosystem. Such redundancy should serve to make smart contracts, and DeFi as a whole, less vulnerable to exploits from implementation bugs.

We hope that this paper can offer one step on the path to a more robust and secure DeFi. As in avionics, in DeFi, the stakes are high, and the risks real.

References

[1] CryptoSec, "Comprehensive list of defi hacks and exploits," Nov 3, 2021. [Online]. Available: https://cryptosec.info/defi-hacks/

[2] N. Mutual, "A decentralized alternative to insurance," Nov 3, 2021. [Online]. Available: https://nexusmutual.io/

[3] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996, pp. 293–307 vol.1.

[4] Ethereum, "Ethereum nodes and clients," 2021. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/

[5] ——, "Layer 2 rollups," 2021. [Online]. Available: https://ethereum.org/en/developers/docs/scaling/layer-2-rollups/

[6] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, "Sok: Off the chain transactions," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 360, 2019.

[7] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1325–1341. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[9] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (defi)," *CoRR*, vol. abs/2101.08778, 2021. [Online]. Available: https://arxiv.org/abs/2101.08778

[10] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of 26th Annual Network & Distributed System Security Symposium (NDSS)*, February 2019. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/111410/

[11] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the defi ecosystem with flash loans for fun and profit," 2020.

[12] Bloomberg, "Defi platform mistakenly sends $89 million; ceo begs return," Oct 1, 2021. [Online]. Available: https://www.bloomberg.com/news/articles/2021-10-01/defi-platform-mistakenly-sends-89-million-ceo-begs-its-return

[13] G. Barros and P. Gallagher, "Eip-1822: Universal upgradeable proxy standard (uups)," 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1822

[14] S. Palladino, "Eip-1967: Standard proxy storage slots," 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1967

[15] F. Vogelsteller and V. Buterin, "Eip-20: Erc-20 token standard," 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[16] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[17] V. Buterin, "Endgame," 2021. [Online]. Available: https://vitalik.ca/general/2021/12/06/endgame.html

[18] Uniswap, "Uniswap," 2020. [Online]. Available: https://app.uniswap.org/#/swap

[19] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[20] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[21] Ethereum, "Evm lab utilities," 2019. [Online]. Available: https://github.com/ethereum/evmlab

[22] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "Evmfuzzer: detect evm vulnerabilities via fuzz testing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1110–1114.

[23] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 349–365. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/yang

[24] T. Copeland, "Bug impacting over 50% of ethereum clients leads to fork," 2021. [Online]. Available: https://www.theblockcrypto.com/post/115822/bug-impacting-over-50-of-ethereum-clients-leads-to-fork