This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

Highly Available Blockchain Nodes With N-Version Design

Javier Ron*, César Soto-Valero*, Long Zhang†, Benoit Baudry*, Martin Monperrus*

*KTH Royal Institute of Technology, †Electrolux AB

***Abstract*—As all software, blockchain nodes are exposed to faults in their underlying execution stack. Unstable execution environments can disrupt the availability of blockchain nodes' interfaces, resulting in downtime for users. This paper introduces the concept of N-Version Blockchain nodes. This new type of node relies on simultaneous execution of different implementations of the same blockchain protocol, in the line of Avizienis' N-Version programming vision. We design and implement an N-Version blockchain node prototype in the context of Ethereum, called N-ETH. We show that N-ETH is able to mitigate the effects of unstable execution environments and significantly enhance availability under environment faults. To simulate unstable execution environments, we perform fault injection at the system-call level. Our results show that existing Ethereum node implementations behave asymmetrically under identical instability scenarios. N-ETH leverages this asymmetric behavior available in the diverse implementations of Ethereum nodes to provide increased availability, even under our most aggressive fault-injection strategies. We are the first to validate the relevance of N-Version design in the domain of blockchain infrastructure. From an industrial perspective, our results are of utmost importance for businesses operating blockchain nodes, including Google, ConsenSys, and many other major blockchain companies.**

***Index Terms*—N-Version design, blockchain, availability.**

I. Introduction

Blockchain technology is fundamental to offer secure, reliable, and decentralized software services [1]. Blockchains enable the transaction of digital currencies [2], as well as the creation and execution of smart contracts [3]. Several businesses depend on the correct operation of blockchain networks to provide services to their clients [4]. Major actors such as banks or cryptocurrency exchanges with high volumes of end-users rely on trustworthy and uninterrupted access to said networks [5], [6], [7].

All those actors exercise their mission-critical business on top of blockchain nodes [8]. However, blockchain nodes are never without risk of failure, and blockchain outages have occurred several times, causing downstream service disruptions and loss of revenue [9], [10]. Software failures are often the consequence of operating system, network, or hardware problems, which cause unstable execution environments [11]. Therefore, there is an overt need for techniques that allow blockchain node operators to mitigate the effects of software failures.

N-Version programming is a proven approach to building fault-tolerant systems [12]. N-Version programming consists of creating several implementations called *versions* of a program based on the same specification. These versions are meant to be executed simultaneously with the same inputs, and the produced outputs are compared afterward as means of fault detection and fault tolerance.

For some blockchains, multiple compatible implementations exist, given that blockchains are protocol-driven by design. For example, Ethereum's execution and consensus layers have respectively four and five major implementations [13]. In general, the usage of many versions of blockchain implementations is regarded as an important addition towards achieving systemic reliability [14], [15], [16].

In this paper, our key insight is to take advantage of diverse implementations of blockchain node to enhance dependability properties. This is realized as the novel concept of "N-Version blockchain node", which is an ensemble of diverse blockchain nodes which collectively provide improved services to external clients. We evidence that N-Version blockchain nodes are a valuable approach for users with high-availability requirements. To the best of our knowledge, taking advantage of N-Version design for blockchain is novel and has not been proposed before.

To implement our vision of an N-Version blockchain node, we carry out the following steps. First, we design a novel architecture that provides higher availability to external clients while encapsulating the inner complexity of N-Version software. This N-Version design includes strategies for request routing, error handling, and response comparison and ranking in the context of blockchain nodes. Second, we implement an N-Version blockchain node prototype for one of the most sophisticated, feature-rich, complex, and globally adopted blockchains: Ethereum. We deploy and coordinate several Ethereum client implementations behind a common interface, in production. Third, we set up an original experimental framework for measuring blockchain node availability. This includes characterizing response latency and correctness, designing a comparison oracle in the domain's specific context. For these experiments, we synthesize realistic fault injection strategies that cause unstable execution environments. Last, we compute the availability rates of common blockchain nodes as well as the rates of our N-Version prototype.

Our results provide empirical evidence that the behavior diversity of blockchain clients can be harnessed in an N-Version architecture to provide high availability under an unstable execution environment. Specifically, our prototype provides higher availability than any of the state-of-the-art nodes taken in isolation. Under the tested workloads and fault injection strategies, we observe an increase in full availability

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

```
{
  "jsonrpc": "2.0",
  "method": "eth_getBlockByNumber",
  "id": 1,
  "params": ["0xa55e27", false]
}
```

(a) Ethereum JSON-RPC request.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "difficulty": "0xa9ed0e03a6530",
    "gasLimit": "0xbea427",
    "gasUsed": "0xbe64e3",
    "nonce": "0xde84c6458a7c0aa0",
    "number": "0xa55e27",
    "size": "0x6f75",
    "timestamp": "0x5f5ad163",
    "totalDifficulty": "0x3ab7010902da66c075f",
    "transactions": [...],
    "uncles": [],
    ...
  }
}
```

(b) Partial Ethereum JSON-RPC response.

Fig. 1: Ethereum node and external application interaction.

from 84.7% for the best performing single version, to 98.5% for the N-Version design. Our results also show the trade-offs between availability and CPU, disk, and memory usage.

To sum up, our contributions are:

- The concept of N-Version design for blockchain nodes. A blueprint of an N-Version blockchain node that leverages the natural diversity of blockchain implementations. To our knowledge, this is the first-ever realization of the N-Version design vision in the context of blockchain systems.
- An implementation of the diverse N-Version blueprint for the Ethereum blockchain using state-of-the-art Ethereum implementations.
- A sound methodology for studying availability of blockchain nodes in production using realistic fault injection, and the corresponding sound results demonstrating the strong advantages of N-Version for blockchain.
- A publicly available, open science repository for reproducing our experiments, at https://github.com/ASSERT-KTH/N-ETH/.

II. Background

A. Blockchain Technology

Blockchains are distributed ledgers, where data is aggregated and stored in discrete units called blocks [2]. Blockchains are created by *peer-to-peer networks*, where each peer, or *node*, is a host that executes a blockchain client.

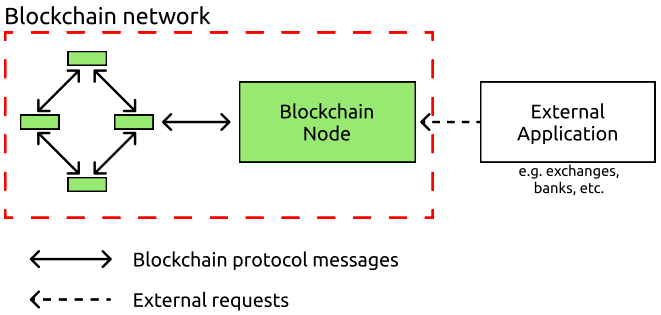


Fig. 2: An external application may interact with a blockchain through any node that exposes its interface.

Blockchain clients implement a protocol, which describes how blocks are verified, agreed upon, and propagated throughout the peer-to-peer network. Decentralization is essential to blockchains, therefore every participating node must be able to store a replica of the ledger, and verify all incoming blocks before making them part of their own state. Ideally, every node also exposes an interface, allowing the blockchain's users to issue read/write queries.

Each blockchain has its own protocol, prime examples of these protocols are Bitcoin's [2] and Ethereum's [17]. All the client implementations for a given blockchain must comply with this blockchain's protocol, and no other restrictions are made on the implementation. This allows for the creation of several, diverse, client implementations for the most popular blockchains [18], [14].

In addition to peer-to-peer communication, blockchain nodes provide standardized outward-facing channels to interact with external applications. For example, Ethereum clients implement the JSON-RPC API specification [19]. This API allows external applications to connect and query the blockchain using a uniform set of methods. The available operations include, e.g. querying the status of the blockchain at an arbitrary point in the past or issuing new, state-altering transactions. External applications take advantage of the API to build various services, such as cryptocurrency exchanges, dApps, or games.

Figure 2 depicts an overview of both a blockchain network and an external application. Here, two types of interactions are highlighted: Dotted lines show external applications requests toward a target blockchain node through its outward-facing interface; and solid lines show the blockchain node which sends and receives data to and from its connected peers.

An example of data exchange between a blockchain node and an external application is shown in Listings 1a and 1b. Listing 1a shows a complete Ethereum JSON-RPC call. It contains the version of the interface (`jsonrpc`), the name of the method to be invoked (`method`), the parameters passed to the method (`params`), and a request identifier (`id`). Listing 1b contains the data returned for the previous request, which is specific to the "`eth_getBlockNumber`" method. Among other information, it contains the number of the block (`number`), its size (`size`), a timestamp (`timestamp`), and a list of transactions (`transactions`).

The example in Figure 2 is greatly simplified compared to reality: a production blockchain network is composed of thousands of nodes spread in complex topologies all over the globe [20]. Each node serves numerous external applications, meaning that there are many more external applications than nodes.

B. N-Version Design

An N-Version software application relies on the simultaneous execution of N programs that are different implementations of the same specification [21]. This type of architecture is used to enhance a desired property of the application, such as reliability [15], [22], performance [23], or security [24], [25]. In most cases, this is achieved by comparing or matching the outputs of the N programs given the same input.

N-Version software applications are typically produced through *N-Version programming*. N-Version programming is defined as the independent development of the same specification by different teams [26]. In this context, each program developed by one team is called a *version*. Ideally, each version is implemented independently using competing designs, programming languages, and software stacks [12]. A high degree of diversity is key, as it lowers the probability of shared faults between versions [27], [28], [29].

Interestingly, there exist software specifications for which multiple isolated implementations surface *naturally* [30]. These implementations emerge with no coordinated effort to produce them. Instead, they emerge spontaneously due to market competition, the need for optimization, or opinionated design approaches [31]. Web browsers are an example of this pattern: they are developed independently of each other by competing actors, and yet they conform to the same standards [32].

When independent versions that comply to the same protocol emerge naturally, it is possible to harness them to build N-Version software applications [33]. This approach can be called *natural N-Version design*.

Definition: Natural N-Version design is the creation of N-Version software applications by harnessing, deploying, and simultaneously executing already-available implementations of a software specification.

III. Highly Available Blockchain Node

A. Blockchain Node Availability

The availability of a blockchain can be defined as the probability that it is functioning correctly at an arbitrary point in time [34]. For the purpose of this paper, we include in this definition the possibility of single blockchain nodes to participate in the network in a degraded state. Therefore, we characterize blockchain nodes' availability as a categorical variable, which can hold three values: *Available:* The API's responses are timely, compliant, and fresh; *Degraded:* The API's responses are not compliant or not fresh; and *Unavailable:* The API's responses are not timely or requests are denied.

The properties of the responses are defined as follows. *Timeliness* refers to obtaining a response to a request within a time span T. For example, if T is set to 100ms, a response time t_r of 150ms is not timely. *Compliance* refers to a response's conformity to the API schema. For example, truncated, non-parsable, or otherwise defective responses are not compliant. *Freshness* refers to how up-to-date the contents of the response are with respect to the global blockchain state. It is measured by computing the distance between the latest block available from the API and an external oracle, namely, another blockchain node. If the freshness value f_r of a response is higher than a maximum freshness value F, we consider the response not fresh.

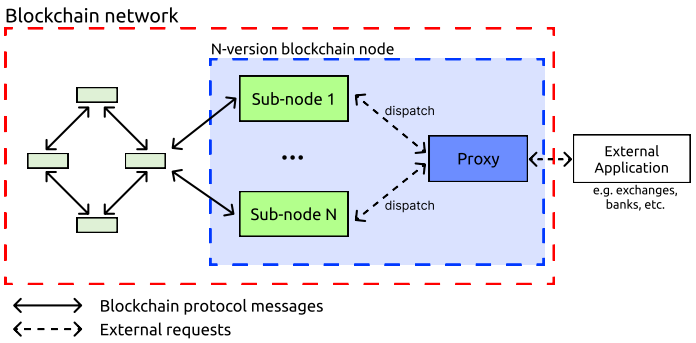


Fig. 3: Concept of an N-Version blockchain node, taking advantage of the presence of multiple versions for the same blockchain network.

We define the availability status S of a blockchain node n with response r as follows:

$$S = \begin{cases} \text{AVAILABLE;} & t_r \leq T \wedge c_r \equiv true \wedge f_r \leq F \\ \text{DEGRADED;} & t_r \leq T \wedge (c_r \equiv false \vee f_r > F) \\ \text{UNAVAILABLE;} & t_r > T \end{cases} \tag{1}$$

Where t_r is the response time, c_r the compliance of the response, and f_r the freshness of the response. We measure t_r as time in milliseconds, c_r as a boolean value, and f_r as the block distance between n and an external oracle. The upper bounds for response time and block distance are T and F, respectively, and are set by the node's operator according to their own requirements.

Given that an individual node response defines the status of the node at one point in time, we consider the node to hold that status until it sends a new response.

B. Architecture of an N-Version Blockchain Node

For some blockchains, there are several compatible client implementations [35]. For example, Ethereum's execution layer has four major implementations. Our goal is to build on this favorable property of blockchains, and apply N-Version design in this context, as shown in Figure 3. We call the resulting construct an *N-Version blockchain node*.

Definition: An N-Version blockchain node is an ensemble of N sub-nodes and a proxy, where a sub-node is a normal node executing a unique client implementation; and the proxy encapsulates the sub-nodes under a single interface.

Our study focuses on assessing the impact of N-Version nodes on availability. The goal is to demonstrate that an N-Version blockchain node provides higher availability than regular nodes in isolation, specially under unstable execution conditions.

1) Overview: Figure 3 shows a blueprint of the proposed N-Version blockchain node. First, it shows the components presented in subsection II-A: Blockchain network, external application, and peer-to-peer communication. The key novelty in our architecture is the proxy component. This component exposes an interface which encapsulates N blockchain sub-nodes, where each sub-node executes a different implementation of the blockchain protocol. Each request directed to the N-Version node must be done through this proxy.

The proxy is responsible for the orchestration of the N-Version node, by routing requests to the sub-nodes depending on a dynamic priority policy and fail-retry mechanisms as explained in subsubsection III-B2. The proxy is also in charge of deciding which response to return to the caller in cases where several responses for a single request are produced (subsubsection III-B3).

2) Dispatching Policy: The presence of sub-nodes is an opportunity to have an adaptive dispatching policy based on their observed behavior. Such a policy allows the system to dynamically adjust to the effects of unstable execution environments, by prioritizing the most available sub-node. This is alike dynamic load balancing, enabling us to achieve system-wide optimization using the global state of the system [36].

The policy works as follows. We keep an availability score for each sub-node. The score is the percentage of successful responses for all requests sent to that sub-node. The score is updated every time a response is received from a sub-node. With this score, we keep a ranking of the sub-nodes. Every time the scores are updated, the ranking is sorted in descending order of availability score. When the proxy receives a request, it is forwarded to the top sub-node in the ranking. If an AVAILABLE response is returned, said response is sent immediately to the requester. Otherwise, the proxy saves the response, and retries the request on the next sub-node of the ranking. This process is repeated until either one of the sub-nodes responds with an AVAILABLE response, or all of the sub-nodes have provided one response.

3) Comparison Oracle: When no sub-node is fully available, there is a need to select the best degraded response to be returned to the external application. For this, the system compares the sub-noed responses and sends back the best one according to the following rules. *Rule 1:* A compliant response is better than a non-compliant response; and *Rule 2:* The most fresh response from all compliant responses is better. Compliance is used as a primary filter, because non-compliant responses can cause undefined downstream behavior. For instance, a response with an incomplete JSON object, could trigger unhandled errors on the external caller.

C. Implementation

We fully implement a prototype based on the blueprint architecture described in subsection III-B in the context of the Ethereum blockchain. We call this prototype N-ETH. We use the readily available implementations of Ethereum execution-layer clients GETH v1.12.2, BESU v23.7.0, ERIGON v2.48.1, and NETHERMIND v1.20.1 as the sub-nodes of the system. Each of the chosen implementations has an active community and is open source. In the rest of this paper, we refer to them as *Ethereum node versions*. The proxy component is written in Go, using networking components from the standard library.

IV. EXPERIMENTAL PROTOCOL

A. Research Questions

To systematically evaluate our architecture and prototype, we propose the following research questions:

RQ1 *What are the behavioral consequences of unstable execution environments for blockchain nodes?*

Blockchain nodes may behave incorrectly due to unstable execution environments. We aim to identify the effects caused by this instability. To this end, we deploy four Ethereum nodes, each with system-call error injection in place. By varying the fault injection strategies, we observe and record a wide range of irregular behavior that may have an impact on availability.

RQ2 *To what extent do different blockchain node versions exhibit different availability rates under unstable execution environments?*

To establish a baseline for availability, we perform a quantitative analysis of the identified effects. To obtain this baseline, we deploy single Ethereum nodes, each with a corresponding system-call error injection module. The baseline consists of precise measurements of the availability state of the nodes under increasingly aggressive fault injection strategies. The availability state is recorded as defined in Equation 1.

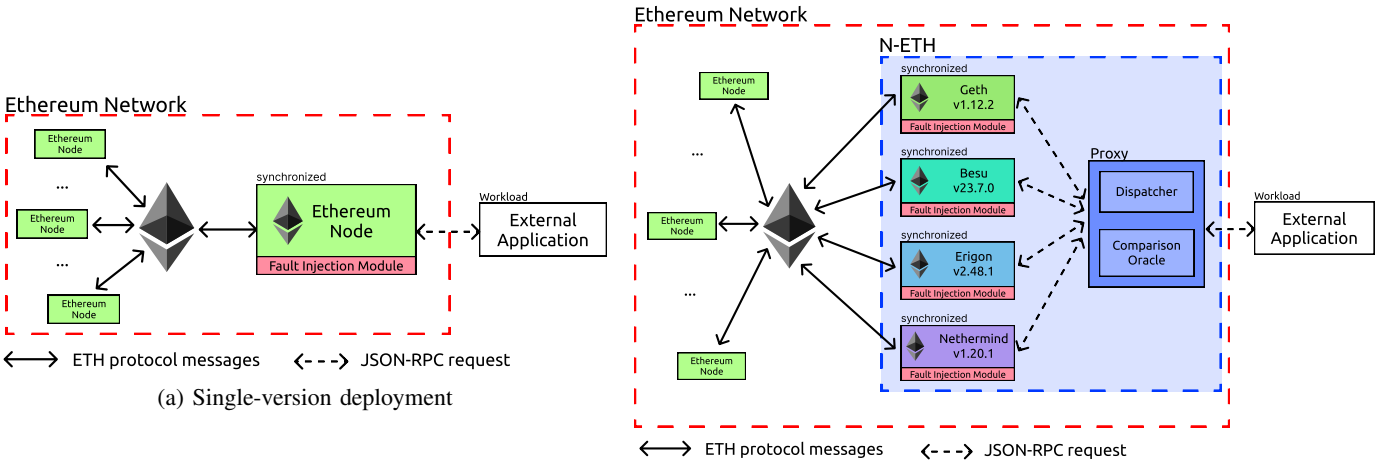
RQ3 *To what extent does an N-version blockchain node increase availability compared to a single-version node?*

We argue that an N-Version blockchain node enhances availability properties of the node under unstable execution environments. To measure this improved availability, we deploy our N-Version Ethereum node prototype with attached system-call error injection modules. We measure its availability score with varying N and compare it against the baseline derived from single-version nodes.

B. Deployments

To answer the research questions, we deploy single-version Ethereum nodes and N-ETH nodes, on which we exert fault injection strategies and workloads. When selecting the versions that constitute the N-Version deployments, we can only select among existing Ethereum node versions. According to the Ethereum community, there are 4 implementations that make up for virtually all participating nodes in the main network [14]. We use these 4 node implementations as

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195



(a) Single-version deployment

(b) N-Version deployment (N=4)

Fig. 4: Overview of the experimental Ethereum deployments. Each node or sub-node takes 8–16 hours to synchronize with Ethereum's Mainnet. In both figures, the Ethereum logo represents Ethereum's Mainnet, which comprises tens of thousands of nodes.

part of our experimental setup: GETH, ERIGON, BESU, and NETHERMIND.

Single-version deployment: Figure 4a shows the scope of the single-version deployment. To realize it, we first deploy a particular Ethereum node version and configure it to synchronize with Ethereum's Mainnet. Once synchronized, we attach a fault injection module to the Ethereum node process, as explained below, in subsection IV-D. The data collected from this deployment provides insights into RQ1 and RQ2, i.e. to identify and measure the effects of unstable execution environments in the nodes' availability.

N-Version deployment: To realize an N-Version deployment, we go through the following steps: First, we create N instances of Ethereum nodes, each coming from a different version. We configure them to synchronize with Ethereum's Mainnet. Second, we deploy an instance of the proxy and connect it with those synchronized nodes as sub-nodes. Third, we attach a fault injection module to each of the sub-nodes. We perform these three steps with N equal to 2, 3, and 4. Figure 4b shows the scope of N-ETH with $N = 4$: the proxy is connected to 4 subnodes, an instance of Geth, an instance of Besu, an instance of Erigon, and an instance of Nethermind. For each value of N, we consider all possible sub-node combinations. In total this adds up to 29 deployments. The data collected from these deployments provides insights into RQ3, i.e. measuring the improved availability under unstable execution environments.

In both single-version and N-Version deployments, we record the availability state for each request, which can take three different values: AVAILABLE, DEGRADED, or UNAVAILABLE, as described in subsection III-A. Additionally, we measure the resource consumption of each deployment, to determine the tradeoff between N and any change of measured availability.

C. Workloads

Workloads are exerted into the deployments through a custom component, which acts as the 'external application' component depicted in Figure 4a and Figure 4b. The workloads consist of an arbitrary number and types of JSON-RPC method invocations targeting the deployment. To quantify the availability of deployment, the workload component records the received responses' conformity, freshness, and latency. We devise two workloads:

Workload A consists of 360 000 JSON-RPC method invocations, where each invocation's method name and parameters are sampled from a pool. This pool contains 21 methods and corresponding parameters, which query both current and past states of the blockchain. The aim of this workload is to discover the widest possible range of availability-related issues induced by our fault injection strategies.

Workload B consists of 360 000 invocations of a single JSON-RPC method, which queries for the latest block available on the target deployment. The aim of this workload is to collect freshness information. It is important to note that while the requests are identical, the responses are expected to change regularly over time as more blocks are added to the chain.

Both workloads are configured to perform each request 5 milliseconds apart. This means that in total, the workloads will perform requests steadily for 30 minutes. Over this time span, the Ethereum blockchain adds 150 blocks to the chain. The selected workload duration and ensuing block distance allow us to observe the effects of unstable execution environments on our deployments.

D. Unstable Environment Simulation

Blockchain nodes are executed on top of an operating system (OS). Consequently, blockchain nodes are susceptible to OS or hardware instability. In environments such as the cloud, single hardware or network faults can propagate to multiple virtual machines [37] affecting multiple blockchain client instances simultaneously. Such instability typically manifests downstream as system-call invocation errors [38]. For

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

example, a `read` system-call may repeatedly fail with error code `-EAGAIN` due to a disk malfunction. Previous research shows that high-frequency system-call errors may cause unexpected behavior [39]. In the context of blockchains, it can result in disruption of block transmission, and chain synchronization [40]. Additionally, permanent side effects and crashes can also be the result of system-call errors. Therefore, we consider system-call errors as the fault model under which we analyze the degradation of blockchain client APIs. Fault models that can make blockchain nodes unavailable with 100% certainty, such as power outages, are out of the scope of this study.

Since fault injection is regarded as an effective way to test N-Version systems [41], we devise several realistic fault injection strategies (FIs) to apply into blockchain nodes. Figure 5 shows the process used to craft realistic fault injection strategies. It consists of the following steps: ① For all the Ethereum node versions used in our deployments, we perform a monitoring procedure, where we record all system-calls and system-call return codes. The return codes reveal system-call invocations which fail. Unsuccessful system-calls are frequent even during correct execution of processes, and may be caused e.g. by temporarily unavailable resources or lost connections. ② We produce a system-call error profile in the form of a set S of tuples with the form $\langle syscall, err, f \rangle$, where $syscall$ is the name of a system-call, err is a system-call return code, and f is the frequency with which $syscall$ returns with code err. ③ We aggregate the sets of each analyzed version into a single set, where no pair of $syscall$ and err is repeated and f is the minimum value between any pair that was repeated. The aggregated set is then sorted in descending order based on f ④ We create n subsets from the aggregated set following a top-n pattern, i.e., subset 1 contains the top-1 tuple from the aggregated set, subset 2 contains the top-2 tuples from the aggregated set, and so on. ⑤ In all tuples of the resulting sets, f is amplified with an arbitrary factor of 5%. We consider this factor to result in balanced scenarios where sporadic errors are likely to be observed, while the relative frequency of system-call errors is kept.

After applying these steps, we obtain 20 fault injection strategies with the following attributes: (1) They are realistic, this means that they only include system-call errors known to occur spontaneously in at least one of the analyzed blockchain nodes; (2) They generate faults uniformly and independent of the deployments' nodes or sub-nodes, and therefore allows comparing resilience between deployments; (3) There is a clear increase of aggressiveness from FI 1 to FI 20. FI 1 contains a single tuple, which is the one with the highest f. We consider the most frequent system-call errors to be handled with high certainty, therefore we regard FI 1 as the least aggressive strategy. On the other hand, FI 20 contains all 20 observed error tuples. This means that FI 20 is the most aggressive strategy, which triggers the highest number of errors, including the most uncommon ones.

To monitor the Ethereum clients' system-calls and to perform fault injection, we rely on the tool ChaosETH [40].

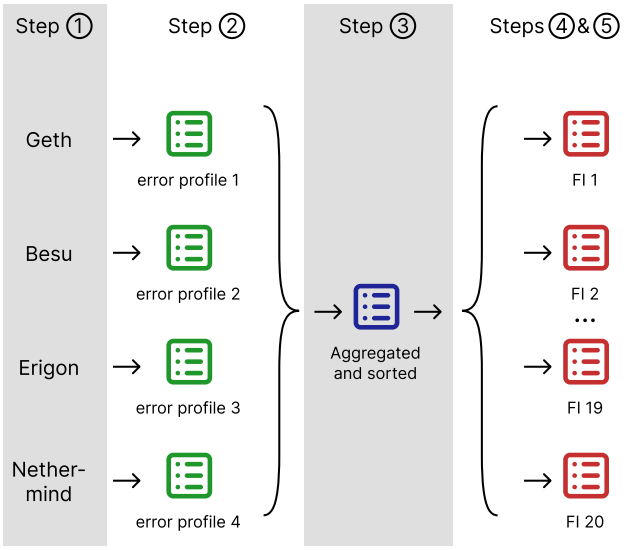


Fig. 5: Fault injection strategy synthesis.

E. Running Experiments at Scale

For our experiments to be the closest to a real-world setting, we require them to be done over the main network of Ethereum, called "Mainnet". Consequently, all nodes must be fully synchronized with Ethereum's Mainnet. However, synchronizing an Ethereum node on Mainnet can be challenging [42].

First, it requires a significant amount of resources: The selected node versions require from 0.8 TBs to 1.2 TBs of space on fast storage devices, as well as between 8 GBs and 16 GBs of RAM. Second, it requires the parallel execution of another kind of blockchain node, known as a *consensus layer* node. Third, synchronizing an Ethereum node from scratch takes from 10+ hours to several days. The time largely depends on the hardware where the node is executed, and the available bandwidth.

We devise experiments as follows: Regarding single-version deployments, we perform one experiment per Ethereum node version (4), per fault injection strategy (20), per workload (2). Regarding the N-Version deployment, we perform one experiment per fault injection strategy (20), per value of N, and corresponding combinations. Setting all experiments ultimately amounts to executing and synchronizing 660 Ethereum node instances.

To handle this scale, we implement a cloud pipeline that allows us to replicate nodes efficiently. The pipeline uses a cloud computing setup which provides access to $n > 1$ SSD devices. The first SSD is reserved for a node instance which is always kept up to date and where no fault injection is performed. We call this instance the *source* node, and it is synchronized from scratch.

The pipeline then continues by executing three asynchronous procedures, as detailed in Algorithm 1. MAIN starts the initial source node synchronization by calling SYNC_SOURCE, waits for the source node to be up-to-date, and finally calls one instance of RUN_EXPERIMENT for each fault injection strategy. We must keep a source node to later

Algorithm 1 Experiment pipeline

```
w ← {A|B}                                   ▷ Workload
d ← {single|neth}                           ▷ Deployment
S ← {S_1...S_20}                            ▷ Fault injection strategies

function MAIN
    SYNC_SOURCE(d)
    wait source_is_synced
    for s ← 0, len(S) do
        RUN_EXPERIMENT(d, w, s)
    end for
    wait all_experiments_finished
    EXIT ()
end function

function SYNC_SOURCE(d)
    while true do
        START_SYNC (d)
        wait copies_in_progress
        STOP_SYNC (d)
        wait not copies_in_progress
    end while
end function

function RUN_EXPERIMENT(d, w, s)
    wait available_ssd
    COPY_STATE()
    START_DEPLOYMENT(d, s)
    RUN_WORKLOAD(w)
end function
```

create copies of its state for use in the subsequent experiments. This means that synchronization from scratch only happens once. SYNC_SOURCE runs an infinite loop, which starts the source node's synchronization, and pauses it while state copying is in progress. This procedure also restarts source node's synchronization if no copying actions are in progress. Stopping the source node's synchronization is necessary, since the state of the node changes constantly with each added block. Keeping the source node constantly synchronized results in the experiments being carried out with the latest production state. Finally, RUN_EXPERIMENT copies the state from the source node's SSD to a newly provisioned SSD, and then starts a deployment which includes the blockchain node, fault injection module, and external application. Then, it starts the experiment's workload. Performing each experiment on a deployment with a clean copy of the deployment's state guarantees that no lingering effects from fault injection are carried from experiment to experiment. This novel pipeline is designed for paralellization, and enables us to carry out the experiments in an acceptable timeframe.

The experiments are executed in Microsoft Azure virtual machines of type L64s v3, each of which provides 64 vCPUs, 512 GBs of RAM, and access to 8x 1.8 TBs NVMe SSDs. This type of environment fits our use case perfectly, since each instance allows us to simultaneously execute up to 8 Ethereum nodes. The estimated total cost of performing the experiments on the Azure platform is 10 000 USD.

V. Experimental Results

A. What are the behavioral consequences of unstable execution environments for blockchain nodes?

Table I shows all the error types received by an external application while the target blockchain nodes are under fault injection. Each row represents the sum of observed errors for all fault-injection strategies, per each single-version deployment, under workload A. We observe that unstable execution environments have different visible effects on blockchain nodes. Specifically, 17 different types of errors surface on the external application, and are related to either network issues, timeouts, or data corruption. Overall, the most frequent type of error is "connect: connection refused". We determine under log analysis that this error arises when a request is directed towards a crashed deployment. On the other hand, data corruption errors such as "malformed HTTP response", or "unexpected end of JSON", happen very rarely and represent only a small fraction of the total error count.

Regarding the distribution of errors, not all error types have the same frequency in every node version, i.e. there are errors which are common in some versions, but rare in others. For example, the error "Post: Client.Timeout while waiting headers", occurs with all versions, however its absolute frequency in each deployment varies by orders of magnitude. Furthermore, the error "invalid character in response" is very frequent in the BESU deployment, but is never triggered in the rest of the deployments. Finally, we observe that within the same deployment, the frequency of errors is not distributed uniformly, and the absolute frequencies of errors range from zero to hundreds of thousands.

Error	GE.	BE.	ER.	NE.
Post: EOF	56	129 713	31	48
Post: Client.Timeout while awaiting headers	4 820	357 864	3 400	45 932
connect: connection refused	170 409	70 623	201 038	534 307
dial tcp: connect: connection reset by peer	-	-	-	1
http: server closed idle connection	-	-	-	2
malformed HTTP response	-	9	-	-
read: connection reset by peer	854	598	6 153	9 263
read: Client.Timeout while awaiting headers	-	-	1	-
Client.Timeout while reading body	3	-	-	1
gzip: invalid checksum	-	865	-	-
invalid byte in chunk length	-	2 599	-	-
invalid character in response	-	293 867	-	-
unexpected EOF	4	2	3	3 373
unexpected end of JSON input	-	1	-	-

TABLE I: Count of error types occurrences when running Workload A under all fault injection strategies. (GE.) GETH, (BE.) BESU, (ER.) ERIGON, (NE.) NETHERMIND.

There are 5 error types in total, which are triggered only in one node version. These observations suggest that the type and prevalence of errors are non-coincidental, meaning that the same injected fault triggers vastly different errors, depending on the node version. This is fully in line with the core N-Version design assumption: diverse implementations exhibit diverse errors.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

	GETH			BESU			ERIGON			NETHERMIND		
RPC method name	FI 19	FI 18	FI 17	FI 19	FI 20	FI 17	FI 20	FI 18	FI 19	FI 20	FI 17	FI 18
eth_blockNumber	0.138	0.184	0.068	0.377	0.022	0.835	0.159	0.144	0.139	0.366	0.377	0.424
eth_estimateGas	0.140	0.182	0.069	0.376	0.023	0.834	0.163	0.147	0.131	0.378	0.377	0.423
eth_feeHistory	0.138	0.183	0.065	0.384	0.023	0.839	0.161	0.141	0.138	0.369	0.372	0.423
eth_gasPrice	0.137	0.182	0.069	0.382	0.024	0.840	0.160	0.143	0.138	0.369	0.373	0.427
eth_getBalance	0.138	0.182	0.066	0.382	0.022	0.837	0.157	0.147	0.136	0.368	0.376	0.421
eth_getBlockByHash	0.136	0.188	0.066	0.392	0.032	0.843	0.158	0.144	0.139	0.375	0.369	0.421
eth_getBlockByNumber	0.131	0.186	0.067	0.390	0.028	0.840	0.161	0.145	0.142	0.373	0.377	0.425
eth_getBlockTransactionCountByHash	0.141	0.188	0.066	0.382	0.023	0.839	0.159	0.142	0.136	0.365	0.376	0.425
eth_getBlockTransactionCountByNumber	0.139	0.185	0.067	0.384	0.022	0.841	0.161	0.146	0.134	0.375	0.368	0.423
eth_getCode	0.140	0.183	0.068	0.389	0.028	0.843	0.157	0.143	0.140	0.367	0.375	0.427
eth_getLogs	0.144	0.187	0.061	0.400	0.039	0.850	0.162	0.147	0.140	0.368	0.381	0.420
eth_getStorageAt	0.139	0.184	0.065	0.379	0.022	0.838	0.156	0.145	0.138	0.368	0.374	0.427
eth_getTransactionByBlockHashAndIndex	0.142	0.188	0.063	0.387	0.022	0.838	0.157	0.150	0.137	0.367	0.371	0.420
eth_getTransactionByBlockNumberAndIndex	0.135	0.184	0.063	0.385	0.023	0.844	0.163	0.144	0.136	0.371	0.370	0.421
eth_getTransactionByHash	0.140	0.188	0.067	0.387	0.026	0.834	0.184	0.166	0.154	0.372	0.374	0.418
eth_getTransactionCount	0.141	0.181	0.067	0.381	0.024	0.838	0.154	0.145	0.136	0.375	0.370	0.419
eth_getTransactionReceipt	0.139	0.179	0.069	0.386	0.024	0.837	0.176	0.161	0.155	0.372	0.379	0.419
eth_getUncleByBlockHashAndIndex	0.143	0.184	0.068	0.391	0.023	0.839	0.157	0.144	0.140	0.374	0.372	0.419
eth_getUncleByBlockNumberAndIndex	0.138	0.182	0.065	0.377	0.024	0.835	0.158	0.140	0.138	0.362	0.379	0.427
eth_getUncleCountByBlockHash	0.139	0.181	0.065	0.382	0.023	0.839	0.156	0.146	0.137	0.366	0.377	0.421
eth_getUncleCountByBlockNumber	0.135	0.185	0.067	0.385	0.022	0.840	0.159	0.144	0.140	0.372	0.373	0.425
SD	0.003	0.003	0.002	0.006	0.004	0.004	0.007	0.006	0.006	0.004	0.003	0.003

TABLE II: Error rate per method, client, and fault injection strategy. Workload A.

Table II shows the proportion of RPC calls which triggered an error in the external application. The first column contains the JSON-RPC method names, and from then on, each column presents the results for each deployment under three fault injection strategies and workload A. For instance, the cell at the intersection of "eth_blockNumber", GETH, and FI 18, indicates that 13.8% of the requests for this RPC-deployment-FI combination cause an error observed in the external application.

The bottom row of Table II shows the standard deviation (SD) for each column. Analyzing the SD values, we find that the effects of the fault injection strategies on the different methods of the APIs are highly uniform. The column with highest SD corresponds to ERIGON under FI 20, with a value of 0.007. This means that the tested fault injection strategies do not have significantly varying effects depending on the measured API method, which is a good indicator of external validity. For simplicity, Table II presents only the results of three fault injection strategies for each node version, the ones with the largest SD. The complete information for all RPC-deployment-FI combinations is available at http://github.com/ASSERT-KTH/N-ETH

Answer to RQ1

Unstable execution environments disrupt the behavior of blockchain nodes in the form of connection issues or broken responses: resets, timeouts, invalid checksums, malformed HTTP or JSON data, etc. These effects depend on the node version, some effects are observed in all versions, while others are version-specific. This validates the core assumption of N-Version blockchain nodes: not all sub-nodes will fail in the same way at the same time in an unstable environment.

B. To what extent do different blockchain node versions exhibit different availability rates under unstable execution environments?

Table III shows the availability rate of all tested node versions while executing Workload B and under all fault injection strategies (FI). Each row corresponds to one FI and the resulting availability rates of each node, the columns correspond to the availability states described in subsection III-A. Regarding full availability, it can be observed that fault injection affects the blockchain nodes in different degrees. There is a pattern where the nodes can handle increasing aggressiveness of the FIs up to a certain point. This pattern is consistent with our way of constructing fault injection strategies by increasing aggressiveness. The first FIs contain the most common system-call errors, and our observations confirm that they are also better handled. The last FIs use rare and potent system-call errors and put more pressure on nodes' availability. Nonetheless, the first noticeable degradation varies between nodes: GETH is able to keep high availability under the first 16 FIs, and NETHERMIND and BESU under the first 6. ERIGON presents a different pattern where full availability is slightly disrupted even by the first FI. Regarding degraded availability, we identify that its main source is the disruption of the deployments' live synchronization under the most aggressive FIs. This results in responses that do not fulfill the freshness property. Regarding full unavailability, we do not identify any global pattern other than correlation to the aggressiveness of the FIs. Additionally, for all node versions, FI 17 causes a sharp increase in unavailability.

The results also capture a diversity of effects on the nodes given the same FI. For example, all clients show contrasting behavior under FI 15: while GETH is almost always available, BESU and NETHERMIND are mostly degraded, and ERIGON shows intermittently available behavior. Overall, the data shows that BESU has the highest availability rate in average, followed by GETH, ERIGON, and NETHERMIND

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

FI Strat.	GETH			BESU			ERIGON			NETHERMIND		
	Available	Degraded	Unavailable	Available	Degraded	Unavailable	Available	Degraded	Unavailable	Available	Degraded	Unavailable
FI 1	☆ 1.0000	0.0000	0.0000	0.9974	0.0025	0.0001	0.9676	0.0324	0.0000	0.9993	0.0000	0.0007
FI 2	☆ 1.0000	0.0000	0.0000	0.9999	0.0000	0.0001	0.9703	0.0297	0.0000	0.9993	0.0000	0.0007
FI 3	☆ 1.0000	0.0000	0.0000	0.9999	0.0000	0.0001	0.9023	0.0977	0.0000	0.9950	0.0041	0.0010
FI 4	☆ 1.0000	0.0000	0.0000	0.9999	0.0000	0.0001	0.9660	0.0340	0.0000	0.9987	0.0000	0.0013
FI 5	☆ 1.0000	0.0000	0.0000	0.9937	0.0062	0.0001	0.9780	0.0220	0.0000	0.9990	0.0000	0.0010
FI 6	☆ 1.0000	0.0000	0.0000	0.9912	0.0087	0.0001	0.9476	0.0524	0.0000	0.9991	0.0000	0.0009
FI 7	☆ 1.0000	0.0000	0.0000	0.9610	0.0269	0.0121	0.9605	0.0395	0.0000	0.0487	0.9392	0.0121
FI 8	☆ 1.0000	0.0000	0.0000	0.9882	0.0000	0.0118	0.9625	0.0375	0.0000	0.0003	0.9875	0.0122
FI 9	☆ 1.0000	0.0000	0.0000	0.9842	0.0038	0.0120	0.0003	0.9997	0.0000	0.0417	0.9462	0.0121
FI 10	☆ 1.0000	0.0000	0.0000	0.9843	0.0037	0.0120	0.0708	0.9292	0.0000	0.0361	0.9515	0.0124
FI 11	☆ 1.0000	0.0000	0.0000	0.9253	0.0573	0.0175	0.0580	0.9420	0.0000	0.0422	0.9453	0.0125
FI 12	☆ 1.0000	0.0000	0.0000	0.9802	0.0024	0.0174	0.0002	0.9998	0.0000	0.0361	0.9519	0.0120
FI 13	☆ 1.0000	0.0000	0.0000	0.9751	0.0076	0.0173	0.1062	0.8938	0.0000	0.0542	0.9339	0.0119
FI 14	☆ 1.0000	0.0000	0.0000	0.9833	0.0000	0.0167	0.0002	0.9998	0.0000	0.1236	0.8638	0.0125
FI 15	☆ 0.9996	0.0000	0.0004	0.9546	0.0277	0.0176	0.2180	0.7797	0.0023	0.0385	0.9488	0.0127
FI 16	☆ 0.9997	0.0000	0.0003	0.9395	0.0427	0.0178	0.0002	0.9994	0.0004	0.0540	0.9338	0.0123
FI 17	0.3627	0.5442	0.0931	☆ 0.6556	0.2482	0.0962	0.6183	0.2911	0.0906	0.4507	0.2602	0.2891
FI 18	0.1374	0.7643	0.0982	☆ 0.6224	0.2851	0.0925	0.5119	0.3698	0.1183	0.1723	0.0883	0.7393
FI 19	0.2628	0.6428	0.0944	☆ 0.6646	0.2308	0.1046	0.5157	0.3778	0.1066	0.0667	0.0210	0.9124
FI 20	0.2102	0.6872	0.1026	☆ 0.6254	0.2887	0.0859	0.5430	0.3371	0.1198	0.4207	0.3316	0.2477
Avg.	0.8486	0.1319	0.0195	☆ 0.9113	0.0621	0.0266	0.5149	0.4632	0.0219	0.3788	0.5053	0.1159

TABLE III: single-version deployments' availability rate for workload B, with varying fault injection (FI) strategies. The node version with the highest availability rate for the FI row is marked with (☆).

respectively. Nonetheless, GETH has the lowest average for unavailability, managing to reply in average to 98.1% (only 1.9% unavailable) of the requests with either available or degraded responses. Furthermore, NETHERMIND significantly underperforms the rest of the nodes when full and degraded availability are combined, under the four most aggressive FIs.

The difference in how the fault injection strategies affect the nodes' availability can be explained by the distinct error handling paradigms of the underlying programming stacks.

Answer to RQ2

The availability of blockchain nodes deteriorates noticeably under unstable execution environments. The measure at which this happens varies depending on the node version and fault injection strategy, in average: GETH's availability drops to 0.8486; BESU's availability drops to 0.9113; ERIGON's availability drops to 0.5149; and NETHERMIND's availability drops to 0.3788. All node versions remain available in certain conditions where the others become unavailable. In other words, none of the tested fault injection strategies make all nodes unavailable simultaneously. Now, we have strong evidence that the available diverse blockchain node versions are suitable for our novel N-Version design.

C. To what extent does an N-version blockchain node increase availability compared to a single-version node?

Table IV shows the availability measurement of our N-Version blockchain node prototype N-ETH under unstable environment while executing workload B. It shows the best performing combinations in average given N values of 2, 3, and 4. With $N = 2$, and executing GETH and ERIGON, N-ETH is able to maintain 94.2% full availability, and 99.9978% combined full and degraded availability. When $N = 3$, with GETH, BESU, and ERIGON, N-ETH is able to maintain 97.3% full availability, and 99.98% combined full and degraded availability. With $N = 4$, and executing GETH, BESU, ERIGON, and NETHERMIND, N-ETH is able to maintain 98.5% full availability, and 99.9999% combined full and degraded availability. Similar to the results presented in subsection V-B, N-ETH is able to perform normally under the first 16 fault injection strategies. For FI 19, the only strategy with imperfect mitigation, only 0.02% of the requests result in an unavailable response.

By comparing N-ETH's rates with single-version's rates, we see that these are equal or improved under all fault injection strategies. This is, the 'available', 'degraded', and 'unavailable' rates are either better than or equal to the best single-version node for all fault injection strategies. More specifically, in the most aggressive 4 FIs, both the 'available' and 'unavailable' rates are strictly better. For instance, in the single-node deployments, BESU is the best at handling injection FI 20, with scores: 0.6254 available, 0.2887 degraded, and unavailable 0.0859. In contrast, N-ETH with $N = 4$ handles the same fault injection strategy with significantly better scores: 0.9210 available, 0.0790 degraded, and unavailable 0. This represents a difference in scores of: $+0.2956$ available, -0.2097 degraded, and -0.0859 unavailable. In summary, this is close to 50% more available.

The 'unavailable' rate of N-ETH drops to zero or close to zero with the increasing N, for all fault injection strategies. This indicates that the N-Version blockchain node sustains at least a degraded service for most of the tested unstable execution environments. It is important to note that the most noticeable gain is achieved for the most aggressive FIs (FI 17-20). For instance, during FI 18, BESU's unavailable score is 0.0925, while N-ETH with N values of 2, 3, and 4 drop to 0.0112, 0.0021, and 0 respectively.

Under fault injection strategies FI 15 and 19, N-ETH with $N = 4$ is slightly worse than N-ETH with $N = 3$.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3346195

FI Strat.	N-ETH-2 (GE. + ER.)			N-ETH-3 (GE. + BE. + ER.)			N-ETH-4		
	Available	Degraded	Unavailable	Available	Degraded	Unavailable	Available	Degraded	Unavailable
FI 1	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 2	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 3	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 4	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 5	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 6	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 7	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 8	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 9	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 10	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 11	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 12	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 13	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 14	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000	1.0000	0.0000	0.0000
FI 15	0.9996	0.0004	0.0000	▲ 1.0000	0.0000	0.0000	0.9999	0.0001	0.0000
FI 16	0.9997	0.0003	0.0000	▲ 1.0000	0.0000	0.0000	▲ 1.0000	0.0000	0.0000
FI 17	0.7377	0.2547	0.0076	0.8424	0.1568	0.0009	▲ 0.9511	0.0489	0.0000
FI 18	0.6042	0.3846	0.0112	0.8587	0.1392	0.0021	▲ 0.9891	0.0109	0.0000
FI 19	0.7626	0.2244	0.0131	▲ 0.9254	0.0745	0.0001	0.8354	0.1645	0.0002
FI 20	0.7383	0.2495	0.0122	0.8470	0.1525	0.0005	▲ 0.9210	0.0790	0.0000
Avg.	0.9421	0.0557	0.0022	0.9737	0.0261	0.0002	▲ 0.9848	0.0152	0.0000

TABLE IV: N-Version node availability for workload B, with varying N and fault injection (FI) strategies. The arrows represent the change of the rates compared to the best single-version node. The table shows only the combinations with the highest availability in average for each N. (▲) indicates the value where the highest gain in availability was achieved.

	Combination	Available + Degraded			Resource usage		
		FI 18	FI 19	FI 20	CPU %	RAM (GBs)	Disk (GBs)
N-ETH N=2	GE. + BE.	0.9797	0.9740	0.9998	338.90	119.07	1850
	GE. + ER.	0.9888	0.9869	0.9878	370.24	355.92	2294
	GE. + NE.	0.9636	0.9616	0.9247	314.49	113.00	2246
	BE. + ER.	0.9994	0.9736	0.9691	269.24	267.51	2350
	BE. + NE.	0.9789	0.9892	0.9435	213.49	24.59	2302
	ER. + NE.	0.9577	0.9351	0.9936	244.84	261.44	2746
N-ETH N=3	GE. + BE. + ER.	0.9979	0.9999	0.9995	489.19	371.25	3247
	GE. + BE. + NE.	0.9997	0.9914	0.9994	433.44	128.33	3199
	GE. + ER. + NE.	1.0000	0.9922	1.0000	464.79	365.18	3643
	BE. + ER. + NE.	0.9995	0.9724	0.9999	363.79	276.77	3699
N-ETH N=4	GE. + BE. + ER. + NE.	1.0000	0.9998	1.0000	583.74	380.51	4596

TABLE V: N-ETH configurations and their respective Available + Degraded scores, and resource usage measurement. (GE.) GETH, (BE.) BESU, (ER.) ERIGON, (NE.) NETHERMIND. This table only presents the scores achieved under the 3 most aggressive FIs. Resource usage is measured under normal execution.

This can be attributed to two main factors. First, the fault injection tool is non-deterministic, i.e. it injects system call errors based on the probabilities defined by the fault injection strategies. Second, N-ETH's sub-nodes are continuously being updated with the latest blocks, i.e. each experiment is carried out using Ethereum's production state. This is a deliberate experiment design decision, as it allows us to measure changes in availability in the current real-world environment. Given these two points, the injected faults across experiments are never performed exactly with the same underlying state. We mitigate this difference across experiments by performing 360 000 requests during each of them.

Table V shows the availability scores of all possible combinations of N-ETH given $N = 2$, 3, and 4. We observe that the increase of availability is correlated to N, and increasing N also increases resource usage, as expected. These results show that there are differences in trade-offs in the combination space. For instance, the combination of GETH and BESU is significantly more available than the combination of GETH and NETHERMIND. In these cases, during the experiment using FI 20, full and degraded combined availability rates are 99.98% and 92.47% respectively. Table V also shows the variability of resource usage from the different combinations. For example, with $N = 2$ the combinations vary by an order of magnitude in terms of RAM. This information is useful for users who want to select sub-node combinations with $N < 4$ for sake of resource constraints or other reasons.

Overall, these experimental results demonstrate that N-Version blockchain nodes provide higher availability than single-version nodes under the same unstable execution environments.

Answer to RQ3

N-Version blockchain nodes have better availability compared to single-version blockchain nodes. Our data shows that the gains are significant, especially for aggressive fault injection scenarios. As compared to single-version blockchain nodes, we observe in average an increase in availability from 84.7%, up to 98.5%. Notably, the $N = 4$ deployment reduces full unavailability to a negligible amount. These results validate the overall usefulness of our novel use of N-Version design in the context of blockchains. Our results are of utmost importance for practitioners who either provide (e.g. Infura) or rely on blockchain nodes (e.g. exchanges, banks, and art platforms).

VI. Discussion

A. Overhead and trade-offs

The main trade-off of N-Version design is the enhancement of a desired property versus increased resource usage. In the case of N-ETH, availability under instability is enhanced significantly at the expense of an increase of computing resources, dictated by the number of versions N, as shown by our results. Yet, access to computing resources is not a major issue for large service providers, since they are already used to run a sizable amount of blockchain node instances [43]. Those providers would greatly benefit from higher availability and automatic resilience to hardware or software-related faults.

B. Threats to validity

Threats to internal validity: We identify two sources of noise that can have an effect on the produced data and corresponding findings. First, we use non-deterministic fault injection strategies, which can trigger system-calls at any time of the experiment, and during different stages of execution. Second, we use fully-synchronized blockchain nodes, which implies that every experiment is performed over a blockchain node that uses a different underlying state. We mitigate both sources of noise by using a large number of requests in each of the experiments.

Threats to external validity: We identify two stages where the obtained data produces generalizable results: First, we generalize the availability metrics obtained from one RPC method (Workload B), to the whole API. We argue that this is realistic, since the effects of unstable execution environments are uniform across the API. External validity would be improved by considering write methods of the API, this is considered as future work. Second, we choose to realize our prototype implementation in Ethereum, as it is a popular, actively supported, and mature blockchain. We argue that our results are generalizable to other blockchain technologies, as they follow generally the same design principles, yet this has to be verified empirically.

VII. Related Work

A. Blockchain Dependability

Kolb et al. [44] survey open challenges regarding blockchain technology, including the need to enhance non-functional properties such as scalability and availability. Weber et al. [45] present a thorough analysis of the availability of major blockchains, and conclude that while read availability is typically high, write availability is low due to uncommitted transactions. While these studies address blockchain availability, they do not account for the effects of unstable execution environments.

The state of client diversity in Ethereum is described by Ranjan [46], and is continuously tracked by the Ethereum community [14]. In the area of dependability through diversity, Garcia et al. [47] present Lazarus, a tool for automatic management of diversity in byzantine fault-tolerant systems. Breidenbach et al. [48] present the Hydra framework, whose goal is to enhance security of smart contracts using N-Version programming. These works study dependability in closely related application domains, however their specific focus is different. Lazarus [47] focuses in systemic-level dependability of BFT systems; and Hydra in dependability of smart contracts. In contrast, this work is focused in the external availability of nodes of blockchain systems.

Regarding security of blockchains, Chen et al. [49] outline the security of the Ethereum ecosystem, by detailing vulnerabilities, attacks, and defenses. Groce et al. [50] invited 23 professional stakeholders to audit Ethereum smart contracts using both tools and manual analysis, with 246 individual defects identified, categorized based on their severity and difficulty. Groce et al. [51] investigate weaknesses in the Bitcoin Core's fuzzing project. While these works address security, a fundamental attribute of blockchain dependability, availability is not their main consideration.

In the work of Li et al. [52], [43], the effects of certain DoS attacks are measured at the node level. Likewise, Yang et al. [15] perform differential fuzzing on Ethereum nodes. Their effort led to the discovery of two consensus bugs in GETH, which greatly contributed to strengthening the Ethereum blockchain. However, these works' focus is different from ours, namely security and consensus reliability.

B. Software Diversity

Multi-version approaches such as N-Version programming and N-variant systems have been extensively researched and proven to enhance security and fault tolerance [30]. Seminal work from Avizienis et al. [12] introduced N-Version programming, which highlights the opportunities of diverse computation for making fault-tolerant systems. Theoretical analyses and models of N-Version software [53], [54] agree that independence of behavior is crucial for achieving fault-tolerance goals. These works also agree that in practice this independence cannot be guaranteed, even if the versions are developed by distinct teams or using distinct methodologies [55]. Therefore, the applicability of N-Version software has been empirically studied over an expansive range of domains. Within this range we highlight the works of Xue et al. on web browsers [32], Chauvel et al. on cloud architecture [56], Huang et al. on web services [57], Oberheide et al. on anti-viruses [58], and Harrand et al. on Java libraries [59], because of their use of natural diversity, as opposed to planned

diversity in Avizienis' vision. Similar to N-ETH, the mentioned works leverage domain knowledge to achieve fault-tolerance, availability, and security. However, to the best of our knowledge, this is the first work to propose natural diversity and N-Version design as a means of hardening blockchain infrastructure.

N-Version programming traditionally relies on majority voting to select a response to a request; however, several alternatives have been presented. For example, Vouk et al. [60] proposes consensus voting, where a response in an N-Version system is selected only after $M < N/2$ versions agree to accept a response. Going beyond voting, Gao et al. [61] describe the use of Hidden Markov Models to compute the low-level behavioral distance of versions for anomaly detection. In our work, we propose selecting a final response after only one timely, compliant, and fresh response is produced from any subnode. This approach allows N-ETH to provide high availability.

A similar, but more security-focused concept is N-variant systems [62], [63]. In contrast to N-Version programming, "variants" are automatically generated. Koning and colleagues propose MVARMOR, an N-variant execution engine that exploits hardware virtualization to detect divergent behavior among program variants [64], where behavior divergence is observed at the level of system-calls. Voulimeneas et al. [65] show how N-variant execution can be based on the diversity of Instruction Set Architectures by running programs natively in different machines. Berger et al. [66] propose a runtime system to handle errors through diversified memory layouts. Polinsky et al. [67] propose an extension to N-M-variant systems, where M represents a number of replicas for each variant and guarantees a constant N throughout a period where a variant's instance might be unavailable. N-variant systems increase diversity at low levels in single software stacks. As such, they do not mitigate flaws originating from application design, dependencies, or programming languages, which is N-ETH's explicit goal. Nonetheless, the mentioned N-variant approaches and N-ETH are not mutually exclusive. These can potentially be used in combination, providing an even greater spectrum for resilience by diversity.

The proxy pattern and N-Version design are often studied in combination. For instance, Espinoza et al. [25] describe a design where N-Versioned microservices are placed in between proxies, allowing the system to compare both upstream and downstream request-response pairs. Simillarly, Durieux et al. [68] leverage protocol diversity through an HTTP proxy to introduce self-healing for HTML and JavaScript code. These works show that diversity together with proxies can be used to augment targeted aspects of software, e.g. mitigating security concerns or providing automatic code repair. In the intersection of blockchains and N-Version design, proxies are addressed in smart contract resilience. For instance, The Hydra framework [48], describes the entry point of their N-Version smart contracts as a "generic proxy" which delegates incoming transactions to each version. Similarly, Péter et al. [69] propose a proxy between N-Versioned smart contracts and the underlying storage of the Hyperledger Fabric blockchain. These works show that the proxy pattern can be applied at distinct layers of blockchain systems. However, none suits N-ETH's problem statement: high availability in unstable environments. These previous works are different from N-ETH's original solution of RPC routing, response selection and adaptive proxying tuned for blockchain nodes.

VIII. Conclusion and Future Work

In this paper, we identify the potential of taking advantage of existing diversity of blockchain node implementations. We devise an architecture that aims to improve the availability of blockchain clients under suboptimal, unstable execution. We implement a prototype based on this design: N-ETH, and evaluate its availability against regular blockchain nodes. To simulate unstable execution environments, we use a system-call error injection tool.

Our findings show that: (1) External applications which consume blockchain nodes' APIs perceive erratic behavior when the target node is under unstable execution environments; (2) The availability of blockchain nodes is affected by the tested unstable execution environments. The severity of the effects in availability scales with the aggressiveness of the used fault injection strategies; and (3) The N-Version blockchain node prototype is able to stay in available or degraded state under most of the tested unstable execution environments. Additionally, N-ETH presents a drastic reduction in unavailability when compared to common blockchain nodes, which present much larger unavailability windows under the same unstable execution environments. Ultimately, this is the benefit of relying on strong versions that have different weaknesses, as N-ETH mitigates the failures surfacing on specific node version and fault scenario combinations.

In the presented architecture and prototype, we focus on blockchain node implementation diversity. However, we can identify two other dimensions where diversity is relevant and applicable. First, *operating system diversity*, where blockchain nodes are executed on top of diverse operating systems. This approach has the potential to enhance OS-related fault tolerance and security. Second, *single node diversity*, where different versions of the same node are used to detect regressions or errors introduced in newer versions.

In this paper, we focus on improving on availability, which is business critical to external clients and applications. Yet, we envision that N-Version blockchain nodes can enhance dependability attributes other than availability, such as reliability and security. Furthermore, it can be used to enhance performance metrics perceived by external clients such as latency and throughput, given that different blockchain nodes are based on competing design principles.

References

[1] A.M. Antonopoulos, G. Wood, and G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018.
[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[3] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.
[4] Yan Chen and Cristiano Bellavitis. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, 13:e00151, 2020.

[5] Luisanna Cocco, Andrea Pinna, and Michele Marchesi. Banking on blockchain: Costs savings thanks to the blockchain technology. *Future Internet*, 9(3), 2017.

[6] CoinGecko. Top crypto exchanges ranked by trust score. https://www.coingecko.com/en/exchanges.

[7] César Soto-Valero, Martin Monperrus, and Benoit Baudry. The multibillion dollar software supply chain of ethereum. *Computer*, 55(10):26–34, 2022.

[8] Joe Abou Jaoude and Raafat George Saade. Blockchain applications – usage in different domains. *IEEE Access*, 7:45360–45381, 2019.

[9] Scott Chipolina. Crucial ethereum service infura suffers major outage. https://decrypt.co/47846/ethereum-backbone-infura-suffers-major-damage, November 11th 2020.

[10] Andrew Hayward. Metamask, ethereum apps down as infura suffers outage. https://decrypt.co/98457/metamask-ethereum-apps-down-infura-outage, April 22nd 2022.

[11] Xiaolin Teng, Hoang Pham, and Daniel R. Jeske. Reliability modeling of hardware and software interactions, and its applications. *IEEE Transactions on Reliability*, 55(4):571–577, 2006.

[12] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[13] Ethereum Community. Ethereum docs: Nodes and clients. https://ethereum.org/en/developers/docs/nodes-and-clients/, 2021. Accessed: 2021-08-10.

[14] Ether Alpha. Ethereum client diversity. https://clientdiversity.org/. Accessed: 2022-12-05.

[15] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 349–365. USENIX Association, July 2021.

[16] Etherscan Blog. Battle-testing ethereum's finality. https://medium.com/etherscan-blog/battle-testing-ethereums-finality-8909ac1b8ab1, June 2nd 2023.

[17] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger, 2021. Accessed: 2021-04-10.

[18] Bitcoin Wiki. Bitcoin clients. https://en.bitcoin.it/wiki/Clients,.

[19] JSON-RPC Working Group et al. Json-rpc 2.0 specification. *Online]: https://www. jsonrpc. org/specification (Accessed 2021-08-22)*, 2013.

[20] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang, and Xianghong Liu. Toposhot: Uncovering ethereum's network topology leveraging replacement transactions. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 302–319, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[22] Kam S. Tso, Algirdas Avizienis, and John P. J. Kelly. Error recovery in multi-version software. *IFAC Proceedings Volumes*, 19:35–41, 1986.

[23] Sara Gholami, Alireza Goli, Cor-Paul Bezemer, and Hamzeh Khazaei. A framework for satisfying the performance requirements of containerized software systems through multi-versioning. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 150–160, 2020.

[24] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.

[25] Antonio M. Espinoza, Riley Wood, Stephanie Forrest, and Mohit Tiwari. Back to the future: N-versioning of microservices. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*, pages 415–427. IEEE, 2022.

[26] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[27] Laura Zavala and Michael N Huhns. Analysis of coincident failing ensembles in multi-version systems. In *Proc. 19th IEEE International Symposium on Software Reliability Engineering: Dependable Software Engineering Workshop*, 2008.

[28] Jeffrey Voas, Anup Ghosh, Frank Charron, and Lora Kassab. Reducing uncertainty about common-mode failures. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 308–319, 1997.

[29] Paul Townend, Jie Xu, and Malcolm Munro. Building dependable software for critical applications: Multi-version software versus one good version. In *Proceedings Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 103–110. IEEE, 2001.

[30] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48(1):1–26, 2015.

[31] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 333–343. IEEE Press, 2019.

[32] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[33] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 149–159, 2014.

[34] Niclas Kannengießer, Sebastian Lins, Tobias Dehling, and Ali Sunyaev. Trade-offs between distributed ledger technology characteristics. *ACM Comput. Surv.*, 53(2), may 2020.

[35] Collin Adams. The importance of client diversity in decentralized networks, January 17th 2022.

[36] Einollah Jafarnejad Ghomi, Amir Masoud Rahmani, and Nooruldeen Nasih Qader. Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, 88:50–71, 2017.

[37] Mina Sedaghat, Eddie Wadbro, John Wilkes, Sara De Luna, Oleg Seleznjev, and Erik Elmroth. Diehard: Reliable scheduling to survive correlated failures in cloud data centers. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 52–59, 2016.

[38] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.

[39] Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Maximizing error injection realism for chaos engineering with system calls. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[40] Long Zhang, Javier Ron, Benoit Baudry, and Martin Monperrus. Chaos engineering of ethereum blockchain clients. *ACM Distrib. Ledger Technol.*, 2023.

[41] Paul Townend and Jie Xu. Assessing multi-version systems through fault injection. In *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), 7-9 January 2002, San Diego, CA, USA*, pages 105–112. IEEE Computer Society, 2002.

[42] Priyam Keshwa. Ethereum clients' node syncing methods, July 21st 2022.

[43] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. As strong as its weakest link: How to break blockchain dapps at rpc service. In *NDSS*, 2021.

[44] John Kolb, Moustafa AbdelBaky, Randy H. Katz, and David E. Culler. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. *ACM Comput. Surv.*, 53(1), feb 2020.

[45] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–73. IEEE, 2017.

[46] Pooja Ranjan. The state of client diversity in ethereum, August 20th 2020.

[47] Miguel Garcia, Alysson Bessani, and Nuno Neves. Lazarus: Automatic management of diversity in bft systems. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 241–254, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and {Exploit-Resistant} smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1335–1352, 2018.

[49] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[50] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *International Conference on Financial Cryptography and Data Security*, pages 634–653. Springer, 2020.

[51] Alex Groce, Kush Jain, Rijnard van Tonder, Goutamkumar Tulajappa, and Claire Le Goues. Looking for lacunae in bitcoin core's fuzzing efforts. https://agroce.github.io/bitcoin_report.pdf, 2022.

[52] Kai Li, Yibo Wang, and Yuzhe Tang. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1645–1667, New York, NY, USA, 2021. Association for Computing Machinery.

[53] D.E. Eckhardt and L.D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, 1985.

[54] B. Littlewood and D.R. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, 1989.

[55] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986.

[56] Franck Chauvel, Hui Song, and Franck Fleurey. Diversity: A heuristic to improve robustness of self-adaptive cloud architectures. In *Proceedings of the International Conference on Utility and Cloud Computing (UCC)*, pages 132–141, 2015.

[57] Yih Huang and Anup K Ghosh. Introducing diversity and uncertainty to create moving attack surfaces for web services. In *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, pages 131–151. Springer, 2011.

[58] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[59] Nicolas Harrand, Thomas Durieux, David Broman, and Benoit Baudry. Automatic diversity in the software supply chain, 2021.

[60] Mladen A Vouk, David F McAllister, David E Eckhardt, and Kalhee Kim. An empirical evaluation of consensus voting and consensus recovery block reliability in the presence of failure correlation. *Journal of Computer and Software Engineering*, 1(4):367–388, 1993.

[61] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. Beyond output voting: Detecting compromised replicas using hmm-based behavioral distance. *IEEE Trans. Dependable Secur. Comput.*, 6(2):96–110, 2009.

[62] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, 2006.

[63] Michael Franz. Making multivariant programming practical and inexpensive. *IEEE Security Privacy*, 16(3):90–94, 2018.

[64] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 431–442. IEEE, 2016.

[65] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. Distributed heterogeneous n-variant execution. In *Proc. of DIMVA*, volume 12223, pages 217–237. Springer, 2020.

[66] Emery D Berger and Benjamin G Zorn. Diehard: Probabilistic memory safety for unsafe languages. *Acm sigplan notices*, 41(6):158–168, 2006.

[67] Isaac Polinsky, Kyle Martin, William Enck, and Michael K Reiter. nm-variant systems: Adversarial-resistant software rejuvenation for cloud-based web applications. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 235–246, 2020.

[68] Thomas Durieux, Youssef Hamadi, and Martin Monperrus. Fully automated html and javascript rewriting for constructing a self-healing web proxy. *Software Testing, Verification and Reliability*, 30(2):e1731, 2020.

[69] Bertalan Zoltán Péter and Imre Kocsis. N-version programming as a mitigation for smart contract faults in execute-order-validate blockchain systems. In *30th Minisymposium of the Department of Measurement and Information Systems*, pages 33–36. Budapest University of Technology and Economics, 2023.