bBench: A Comprehensive Performance Benchmark for Blockchain Applications

Fernando Richter Vidal, Naghmeh Ivaki, Nuno Laranjeiro
University of Coimbra, CISUC/LASI, DEI, Portugal
fernandovidal@dei.uc.pt, naghmeh@dei.uc.pt, cnl@dei.uc.pt

***Abstract*—The performance assessment of blockchain applications holds significant challenges due to their decentralized architecture, immutable smart contracts, distributed ledgers, and operational costs such as gas fees. Existing blockchain benchmarks often either fail to fully capture blockchain-specific behaviors or offer limited configurability and metric reporting. In this paper, we present a new and comprehensive benchmark designed explicitly for blockchain applications, named bBench. Building on established principles from traditional benchmarking and by specializing them in the blockchain context and supported by customized blockchain tools (i.e., Hyperledger Caliper, web3.eth, and node-os-utils), bBench characterizes blockchain application performance in four dimensions: network performance, resource utilization, storage usage, and operational cost. We demonstrate the effectiveness of our benchmark through a case study involving 12 smart contract applications with varying performance demands, some of which hold known vulnerabilities. The results show the benchmark's ability to quantify performance deviations across different applications, as well as those caused by the activation of specific vulnerabilities.**

***Index Terms*—Blockchain, Blockchain Applications, Smart Contracts, Performance Evaluation, Benchmarking, Metrics**

I. INTRODUCTION

Blockchain applications have specific characteristics that make them distinct from traditional applications. These characteristics include the highly decentralized and complex nature of the entire system, the immutability of data generated by smart contracts, the distributed nature of the ledger where this data is stored, and the costs associated with running a blockchain application (e.g., gas fees or the effective cost of executing a transaction) [1], [2]. These make it difficult to fully assess the performance of a blockchain application and to verify key aspects, such as whether throughput is acceptable, whether there are performance bottlenecks, or whether the operational costs are too high.

A benchmark is a standardized process used to assess how well a system performs with respect to a specific property, such as performance, dependability, or resilience [3]–[5]. Benchmark results enable the assessment of a single system as well as the comparison of different alternatives, such as various database systems [6], different virtualization technologies [7], or different high-performance computing systems [8], among others. Organizations like the Standard Performance Evaluation Corporation (SPEC - spec.org) or the Transaction Processing Council (TPC - tpc.org) have a long-standing tradition in performance benchmarking across numerous domains, including Online Transaction Processing Systems (OLTP), the Internet of Things (IoT), Big Data, and AI. At the time of writing, no benchmark for blockchain applications has been proposed by any of these organizations. Since a smart contract cannot really be changed after it is deployed, having a practical way to assess the performance of a blockchain application before deployment is of key importance.

Previous research has explored the field of benchmarking blockchain applications. Some of the research in this field focuses on permissioned networks or offers limited sets of metrics [9], [10]. Some works are based on general network emulators and lack the ability to allow finer control over blockchain-specific behaviors such as transaction cost dynamics and contract size variability [10]. Works like Diablo [1] and BlockCompass [11] are quite complete but lack support for workload configurability and also do not focus on some metrics that are important when the goal is to provide a broader assessment that includes not only the more usual network-level metrics, but also other types of metrics, such as the transactional cost, block size, or state storage size.

In this paper, we propose a new performance benchmark, named bBench, for blockchain applications. The primary objective is to provide users with a means to assess and compare the performance of different smart contract implementations. Our approach builds on general yet well-established concepts from performance and dependability benchmarking [3], [4], such as workload definition and metrics, while incorporating the necessary adaptations to accommodate the specificities of blockchain (e.g., the existence of gas and the utilization of ledger space). We also customized state-of-the-art blockchain tools (e.g., we modified Hyperledger Caliper to collect fine-grained data) for network emulation, contract deployment, and performance measurement. The resulting benchmark allows the evaluation of a blockchain application's performance across four key dimensions: network behavior, storage usage, computational resource consumption, and operational cost. Each dimension includes selected metrics, some of which were identified in the literature, while others are new and derived from the tools integrated into our benchmark.

We conducted a case study involving a total of 12 smart contract applications. These include three baseline applications with varying performance demands, each accompanied by three additional versions containing vulnerabilities. In each of these versions, we artificially introduced a known type of vulnerability to understand how effectively our benchmark can assess and distinguish the performance of these applications

with and without vulnerabilities. Results show that the benchmark is able to signal cases of contracts in which, as expected, one or more groups of metrics are affected. The benchmark, along with instructions for executing it, the dataset used in this paper, and the detailed results are available at [12]. Additional resources are available at *https://blockchain.dei.uc.pt*.

The main contributions of this paper are the following:

- A novel comprehensive benchmark, particularly tailored for blockchain applications, targeting four assessment dimensions (i.e., network, resource, storage, and cost) and including various metrics that are essential to extensively characterize systems' performance.
- The improvement of existing blockchain tooling, such as Hyperledger Caliper, which is now used to report additional metrics, as well as the integration of other sources of metrics to allow going beyond typical assessments based on traditional metrics like throughput or latency.
- The demonstration of the benchmark's ability to assess different blockchain applications from distinct domains and quantitatively signal the effect of known vulnerabilities in the overall performance.

The rest of the paper is organized as follows. Section II discusses the state of the art in benchmarking blockchain systems. Section III presents our benchmark proposal, and Section IV presents an experimental case study to demonstrate the capabilities of our benchmark. Section V discusses the results, and Section VI presents the threats to the validity of this work and its limitations. Finally, Section VII concludes this paper.

II. State-of-the-art

A benchmark is a standardized procedure that allows for assessing and comparing certain properties of computer systems, such as performance, dependability, or resilience [3]–[5]. The Standard Performance Evaluation Corporation (SPEC) [13] is a well-established organization that develops and maintains standardized, vendor-agnostic benchmarks and tools for assessing the performance of various computing systems. The Transaction Processing Performance Council (TPC) [14] is also a renowned organization that develops and maintains benchmarks for multiple domains, including virtualization, big data, IoT, among many others. Currently, no blockchain benchmark has been made available by any of these organizations.

BLOCKBENCH [9] is, to the best of our knowledge, the first framework designed to assess the performance of permissioned blockchains. The tool is used to assess smart contract execution across different blockchain platforms (i.e., Ethereum, Parity, and Hyperledger Fabric). Results show that Fabric outperforms the other platforms in terms of overall performance, with Ethereum and Parity demonstrating greater resilience to node failures. The evaluation was based on metrics such as throughput, latency, scalability, and fault tolerance. Our intention with bBench is to give it a broader scope in the sense that it applies to permissioned and permissionless networks. As an example, bBench reports the average *gas price*, which is quite important in public environments.

The Boston Blockchain Benchmark [10] is also focused on permissioned networks. The tool is based on Mininet [15], a network emulator designed to replicate realistic network conditions. The benchmark enables the configuration of various network parameters, including topology adjustments, link latency, and bandwidth. The tool reports key performance metrics, including fault tolerance, latency, scalability, and throughput. In addition, the framework also supports the detection of network attacks, such as eclipse attacks. Besides targeting a different set of metrics, our proposal includes a network emulator (i.e., Hardhat) that is specific to blockchain environments, unlike Mininet, which is not designed for blockchain environments. This allows us to emulate much more blockchain-specific behaviors, such as varying network costs, increasing the number of transactions per block, or controlling the size of deployed contracts, to name a few. There are also other solutions for permissioned networks, which share characteristics very similar to those of the previously discussed tools, such as [16] or [17].

Diablo [1] is a blockchain benchmark that comes with five realistic decentralized applications. Diablo's architecture comprises a primary component that coordinates the experiment (e.g., generating and dispatching the workload, aggregating, and reporting results), as well as multiple secondary components responsible for presigning transactions and executing the workload, which involves direct interaction with blockchain nodes. The authors evaluated six blockchain platforms (i.e., Ethereum, Avalanche, Diem, Algorand, Quorum, and Solana). The authors emphasize the importance of using realistic workloads that more accurately reflect the expected behavior of the assessed applications. Diablo offers metrics such as throughput, latency, proportion of committed transactions, peak transaction throughput, and latency distribution over time. Although the work emphasizes the importance of workload in bringing benchmarks closer to realistic settings, it provides limited support for this goal. bBench includes a customizable workload module that supports function dependency handling, and data generation customization using libraries and dictionaries, among other features.

VeriBench [2] allows assessing storage systems that incorporate verifiability mechanisms. The authors use it to evaluate systems comprising blockchains, ledger databases, and log transparency technologies. As a key finding, the authors note that the verification mechanisms implemented (e.g., supported by Merkle trees) have a significant impact on the system's overall performance. Metrics analyzed include latency, throughput, storage usage, abort rate, impact of delay, and impact of block time. VeriBench and our proposal partially share some similarities regarding network-level metrics. However, our proposal introduces different types of metrics (e.g., cost) as well as metrics related to resource and storage usage. It is also worth mentioning that bBench adds various mechanisms to support generating workloads for different applications.

BlockCompass is a benchmark based on Caliper [11]. The authors introduce a workload generation mechanism grounded

in the concept of transactional workload, which emulates multiple users issuing requests in a random manner, including the possibility of failure in some of these interactions. This approach allows for a more realistic evaluation of blockchain applications than those relying solely on static workload batches. The tool is compatible with various networks, including Ethereum, Hyperledger Fabric, and Sawtooth. The key metrics collected by the tool include Emit Rate, Throughput, Error Rate, Latency, Resource Utilization, Scalability, and Gas Consumption. In bBench, we use Hardhat as well as storage metrics that BlockCompass does not support. Furthermore, our workload and cost metrics assess the impact of transfers in payable functions, whereas BlockCompass focuses solely on gas consumption (used to execute the program).

In summary, bBench differs from the state of the art in several aspects. First, it reports a set of metrics that, to the best of our knowledge, is more extensive than those in current works. Second, it offers workload generation capabilities that account for key factors needed to produce valid and realistic workloads (e.g., call dependency handling). Finally, at the implementation level, existing benchmarks such as VeriBench and the Boston Blockchain Benchmark rely on general-purpose emulators. In contrast, bBench utilizes Hardhat, a tool specifically designed for blockchain emulation, which enables finer control over blockchain-specific behaviors, such as transactional costs.

III. A Blockchain Performance Benchmark

In this section, we describe the design of bBench. We first provide an overview of the conceptual design of the benchmark in Section III.A and then explain its implementation in Section III.B.

A. Conceptual Design of the Blockchain benchmark

Our approach is inspired by general models of performance and dependability benchmarking [3], [4], [13], [14], which we have adapted to account for the specificities of the blockchain domain and the context of the target application. In summary, our approach is based on the definition of a *benchmarking scenario*, which sets the basis for the *benchmarking procedure* and includes the following set of main components, detailed in the following paragraphs:

- **Goals:** This component refers to the general goals of the benchmark and mostly reflects the assessment objectives (e.g., overall performance, network, or storage performance). These goals determine which group of metrics should be used or how the workload should be generated.
- **Workload:** This component refers to the combination of inputs to be used in calls to contracts that make up the blockchain application under assessment.
- **Metrics:** This refers to the measurable outcomes (e.g., throughput, memory usage) of the assessment process. It will allow us to measure and understand the effect of running the workload against the blockchain application.
- **Procedure:** This component refers to the overall procedure to be followed during the assessment of the blockchain application. It includes setting up the environment, defining specific rules, and executing the benchmarking process. In general, the procedure involves the following steps: initializing the blockchain system, applying the workload, collecting metrics, and restoring the system's state at specific points in time.

Figure 1 represents the conceptual design of the blockchain benchmark and shows how the different benchmark components (represented in blue rounded squares) interact.

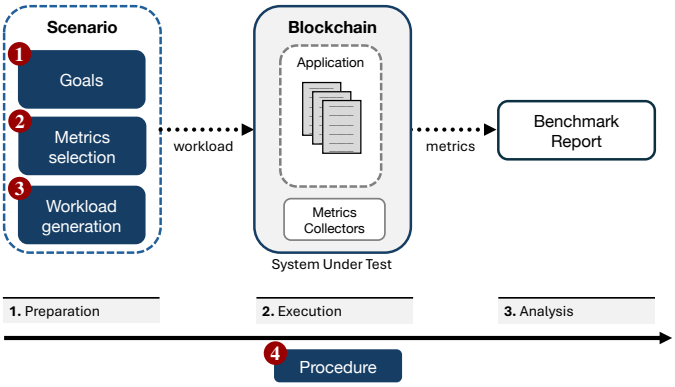

Fig. 1. Conceptual design of the blockchain benchmark.

The benchmarking process begins with the definition of the *benchmarking scenario*, which consists of setting goals, specifying metrics, and configuring the workload generation. The **goals** ❶ may be general, i.e., all available metrics should be used or may signal that the benchmark user is interested in evaluating specific aspects of the system, such as assessing network performance, storage usage, resource consumption, or operational cost. This represents a general indication that specific groups of metrics should be used. Goals may also be used to generally specify some semantics of the application (e.g., what certain operations to assess), which will later support the generation of a specific workload.

The **metrics selection** ❷ component refers to measurable aspects of the blockchain system. During our analysis of the state of the art, we identified the following groups of metrics (see further details in Section III-B and Table I):

i) Network: This group includes typical metrics in distributed systems, such as latency and throughput, which reflect network performance.
ii) Storage: This group refers to metrics such as block size or the number of committed transactions, which reflect the application's performance in terms of storage usage.
iii) Resource: This group includes metrics that refer to the usage of system-level resources, such as CPU or memory usage.
iv) Cost: This group includes metrics related to the cost of running the application and transactions, specifically gas prices and execution cost.

Network metrics groups aspects related to network performance. Some types of applications, such as Decentralized

IoT, can be more sensitive to latency, as delays may render an operation useless, requiring a rollback of an entire IoT process [18]. Other types of applications may be sensitive to different groups of network metrics, such as throughput, which is necessary for applications requiring high data transfer rates, such as Blockchain-based streaming [19], file transfers [20], and web3 gaming [21], to name a few.

Storage metrics are relevant because, in a blockchain architecture such as the one used by Bitcoin, the 80 bytes available for data storage per transaction [22] are insufficient. Combining multiple registers to obtain more space is extremely expensive. Additionally, recording unwanted data harms the network, as it increases the blockchain size, which may discourage nodes from maintaining large copies on their devices. For these reasons, solutions that store external data off-chain have emerged [23], allowing the storage of only a reference link in the ledger. So, the number of affected transactions, the size of the data recorded in the transaction, or the size of the event log stored are relevant for measuring the effect of running a blockchain application.

Resource metrics mostly concern system-level resources, namely CPU and memory usage. Certain applications are more memory-intensive (e.g., operations such as retrieving customer profiles [24]), while others are more CPU demanding (e.g., opening accounts and processing transfers between accounts [25]). From the assessment point of view, it is important to characterize how well the application uses these resources.

Cost metrics (or financial metrics) enable the assessment of how well the blockchain system performs in terms of aspects with financial implications. The increasing number of cyber-attacks targeting cryptocurrency wallets [26], along with the many existing verification tools specialized in related vulnerabilities [27], motivate the need for this group of metrics. A benchmark concerned with the financial impact of a blockchain application should report metrics such as execution cost (Ether balance), which reflects the total cost of executing a given set of operations on the blockchain system.

The next component shown in Figure 1 refers to the **workload generation** ❸. Three main types of workload may be used: real, realistic, and synthetic workloads. The selection of one of these types may depend on the availability of data and on the specific goals of the assessment. The different types of workload can be defined as follows:

1) Real workloads are obtained from a real blockchain system and collected during operation, reflecting the real operational usage, including the number of users and requests that the system should handle. This data is obviously very system-specific and is very often unavailable.
2) Realistic workloads are artificially designed to resemble typical usage of the system while adapting to respect the experimental environment, including specific constraints that may exist (e.g., limited computing resources availability). These workloads can be further adapted to meet specific experimental goals.
3) Synthetic workloads are fully built based on predefined parameters, which allows them to be used in a controlled manner. These workloads are flexible in the sense that they can be used in quite different scenarios, but they may not really represent real operational conditions.

As previously mentioned, the *goals* component will provide information that is used for workload generation. In many cases, for simplicity, the workload will be synthetic, but in some cases, the workload must be somewhat realistic so that the measurements are meaningful. So, besides basic aspects like the smart contract functions to be called, more complex aspects such as setting a specific execution order for the contract operations, signaling dependencies between operations, or the way specific argument values should be generated, are extremely helpful so that the workload complies with minimal coverage and business requirements. For example, in an application that manages hotel reservations, a room must be reserved before items and services can be consumed. Checkout can only be executed after the guest's stay is concluded. This kind of scenario is very frequent in smart contract applications [28] and must be modeled by the benchmark.

The benchmark **procedure** ❹ involves collecting the inputs from the scenario and metrics information and taking care of starting, running, and terminating the benchmark according to the configuration provided by the user. In particular, besides determining the setup through which the benchmark runs, it sets the points where the system should be restored (if any).

B. Benchmark implementation

This section complements the conceptual description of our benchmark by specifying its concrete implementation and detailing the components described in the previous subsection. The benchmark is permanently available for download at [12]. We begin by visually representing the concrete benchmark environment in Figure 2.

To support the definition of **goals**, our benchmark allows the definition of concrete rules, which are primarily used to select an appropriate group of metrics or to guide the type of workload generation (e.g., in some cases, purely random generation might be enough). The definition of *goals* may also determine whether the entire set of metrics should be considered or if only a subset is relevant for the benchmark's output. For instance, a user may have the goal to assess a certain latency-sensitive application and disregard or give less importance to other metrics or may be interested in understanding if a certain contract allows for high transactional throughput or in understanding its behavior in terms of blockchain storage. The specific way how such metrics should be exactly combined is something that is out of the scope of this paper and could be achieved using, for instance, multi-criteria decision-making techniques that would fit complex scenarios.

As previously mentioned, the **metrics** provided by our benchmark fall into four groups: network, resource, storage, and cost. These groups include generic metrics, such as throughput and memory usage, as well as blockchain-specific metrics, such as gas price and execution cost. They also cover even more specialized blockchain metrics, such as block size and state size. To identify the set of metrics that

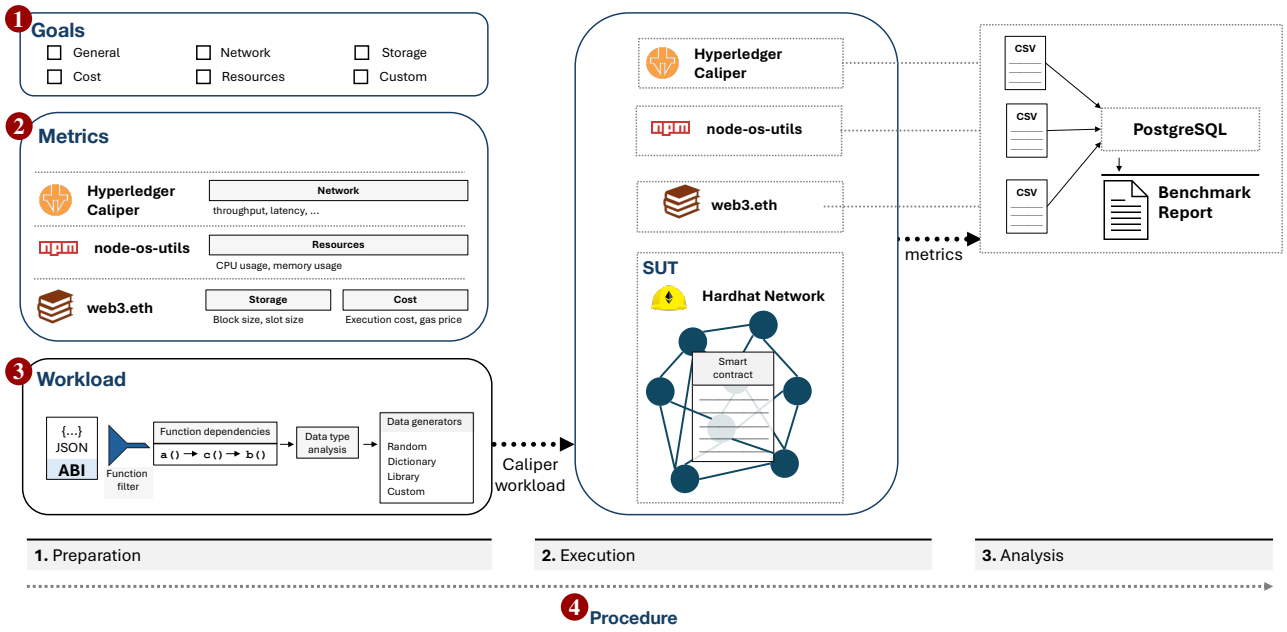

Fig. 2. General view of our benchmark implementation.

could be the target of our benchmark, we used the Google Scholar engine along with the query "blockchain benchmark" to search for papers in which authors used some form of blockchain assessment, snowballing [29] was also used during the process. From each paper, we identified which metrics were used, consolidated the names, and grouped them. We excluded some metrics such as phantom read [30], invalid item [30], concurrent update conflict [30], validationQuorum [31], loadFactor [31] due to being specific of certain blockchains (e.g., phantom read is specific of Hyperledger and validationQuorum is specific of XRP). Table I presents the list of metrics currently supported by our benchmark.

The whole set of metrics is then collected through three tools/sources. The first tool we use to retrieve metrics is Caliper, which is a performance measurement tool. Caliper allows us to collect typical values used in performance measurement (e.g., latency, throughput, etc.), but the source code was also modified to include additional fields, such as network behavior, round ID, error messages, and execution date and time, which are considered essential for the consolidation and statistical analysis of the data gathered from all metrics sources. The second tool is web3.eth, which we use to retrieve additional data related to the blockchain, such as balance, transaction count, block size, storage, and events. The last tool is node-os-utils, which is an operating system tool that retrieves system-level data, such as CPU and memory usage.

The **workload** component is implemented based on the information available on the Application Binary Interface (ABI) code, which contains all contract definitions (i.e., functions, public variables, and events) that can be accessed externally. The ABI also provides argument types (e.g., int, address) and state mutability information (e.g., view, pure), indicating whether the function modifies the contract's state. We also rely on such function attributes to check if a function is payable, so we can automatically generate valid Ether transactions.

In the simplest form of workload generation, the ABI is processed, and data is randomly generated according to each identified data type, e.g., random strings for string parameters and random integer values for integer parameters. If a function is specified as payable (i.e., it can receive Ether), a random monetary transfer is also generated for it. All identified functions are scheduled for execution in a non-deterministic sequence. This can then become more complex by allowing the user to indicate (currently in a JSON configuration file): i) specific functions that should be used for workload generation (i.e., a subset of those present in the contract); ii) the enforcement of a specific order between certain function calls; iii) the maximum and minimum number of times each function should be executed; iv) whether generation is random or constant values should be used for certain parameters; v) the address of the input arguments to be used as msg.sender; vi) the amount of value (in wei) to be sent with each call to payable functions; vii) the gas limit allowed during the execution; viii) the name of custom functions used to generate values for special parameters (i.e., custom functions can be implemented in a `customFunction.js` file); ix) independent execution units that run performance test workloads in parallel; x) whether the state should be restored and when (i.e., so that the effects of previous runs do not accumulate); and xi) specific business rules that must be respected (e.g., special domain values that a workload should include, special moments in which a particular value should be used).

In addition to random generation, the benchmark also uses third-party libraries for certain cases, such as email addresses

TABLE I
METRICS TRACKED AND REPORTED BY BBENCH.

Group	Metric	Unit	Formula	Description	Source	Reference
Network	Throughput	Transactions commited per second	committed transactions / period in seconds	Refers to the rate at which the blockchain commits valid transactions, not at a single node but at all network nodes given a certain period.	Caliper	*throughput*: Duan et al. (2020) [10]; Dinh et al. (2023)[9]. *peak transaction throughput*: Gramoli et al. (2023)[1]; Nasrulin et al. (2022) [17]
	Latency	Miliseconds	sum(latency-individual) / number of committed transactions	Refers to the average time taken by all executed transactions in an experimental run, measured from the moment a transaction is issued to when a response is received.	Caliper	*latency distribution over time*: Gramoli et al. [1](2023). *average latency:* Gramoli et al. [1] (2023). *latency:* Duan et al. (2020) [10]; Dinh et al. (2023) [9]; Yue et al. (2023) [2]. *serverLatency:* Touloupou et al. (2022) [31]
	Committed Transactions	Percentual	committed transactions / total transactions	Indicates the success rate of executed transactions.	Caliper	*emit rate:* Rasolroveicy et al. (2024) [11]. *commit timeouts:* Klenik et al (2022) [30]
	Committed Consensus	Percentual	committed transactions / verified transactions (validated by consensus mechanisms)	Ratio between the number of committed transactions recorded in the ledger and the number of transactions verified by the consensus mechanism.	Web3.Eth	*proportion of commited:* Gramoli et al. (2023) [1]. *endorsement timeouts:* Klenik et al (2022) [30].
Resource	CPU Usage	Percentual	avg(cpu usage) per individual transaction	Refers to CPU usage in percentage, given a reading interval	node-os-utils	*resource utilization:* Rasolroveicy et al. (2024) [11].
	Memory Usage	MegaByte	Sum(mem_end - mem_start) per individual transaction	System memory used, in average, per each transaction. Considers the period starting from the beginning of the transaction to its end (be it successful or not)	node-os-utils	*consumption:* Saingre et al (2020) [16]
Storage	Block Size	Bytes	avg (block sizes generated in the experiment) per individual transaction	Size of the block(s) generated per each transaction, in average.	Web3.Eth	—
	State Size	Bytes	avg(memory used by all declared variables in the contract) per individual transaction	Average of the state storage used by all transactions.	Web3.Eth	storage usage: Yue et al. (2023) [2];
Cost	Gas Price	Wei	avg(gas price) per individual transaction	Amount the user wants to pay per gas unit as a fee to the miner. The higher the rate, the faster the transaction is executed.	Web3.Eth	*gas consumption:* Rasolroveicy et al. (2024) [11].
	Execution Cost	Ether	sum (balance_end - balance_start) per individual transaction	Sums the total cost of each executed transaction	Web3.Eth	—

[32], zip codes [33], or phone numbers [34]. For example, in a farming application requiring geographic coordinates [35], if a random generator is used for latitude and longitude, many of the generated values will be invalid. Finally, the benchmark also makes use of dictionaries to generate realistic data such as city names, first names, surnames, and more. The main idea is that, if needed, the benchmark can generate workloads that better represent real-world execution. As output of this workload generation step, the benchmark produces an execution script with all calls to perform and corresponding generated data.

The *procedure* component of the benchmark, in this context, operates by collecting the overall inputs from the *scenario* and managing the benchmark's start, execution, and termination according to the user-provided configuration. The *procedure* may involve setting the workload script to run once or may consider a certain amount of time to execute each run (each benchmark run is composed of several calls to the contract functions). If the generated calls are insufficient to complete the time set by the user, the workload script will restart from the beginning once it reaches the end, and the state will be reset. The procedure should also specify if the state should be restored after each run or not. Figure 3 presents the execution profile of a set of benchmark runs.

The **System Under Test** includes the infrastructure and an application (i.e., one or more smart contracts). In terms of infrastructure, our benchmark includes a set of software to allow the execution of blockchain applications in realistic environments while collecting all necessary metric values. To support the application, we use Hardhat [36], which supplies the necessary environment and mechanisms to emulate a local Ethereum blockchain. It uses less disk storage than other alternatives, such as the Ethereum TestNet, and has been used previously in similar research contexts [37], [38], [39]. Hardhat also allows emulating useful network behaviors, such as "busy" (i.e., when a high number of calls to contracts are being executed) or "miner activity" (i.e., manipulating the block creation operation by defining the number of transactions that can be stored per block; the smaller the number of transactions, the more blocks need to be generated). In terms of configuration, it is possible to use Hardhat to i) set the maximum gas allowed per block; ii) set the gas limit per transaction; iii) set the gas price in wei; iv) set the minimum allowed gas price; v) bypass contract size limits; and vi) set custom accounts with specific private keys and balances.

Hyperledger Caliper [40] is part of the setup and has the

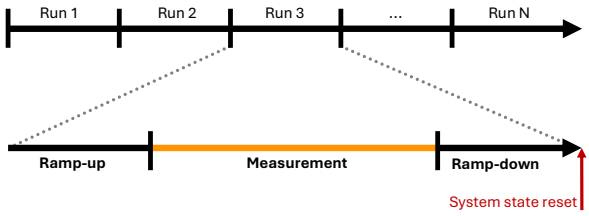

Fig. 3. Benchmark execution temporal profile.

following functions: i) load workloads, ii) execute operations, iii) interact with the blockchain environment, and iv) generate performance metrics. We use Caliper (version 0.6.0, i.e., the latest at the time of writing) and connect it with the blockchain using the accounts provided by the Hardhat emulator.

The system under test also includes the application, including one or more contracts, that is at the center of the assessment and will be exercised through the generated workload. The benchmark exclusively supports contracts written in Solidity and requires the source code of the contract as input. To generate the ABI, all programs are compiled using Hardhat's built-in compiler.

The **benchmark report** includes values for all collected metrics, previously identified in Table I, along with all data used in their calculation. The individual tools (i.e., Caliper, node-os-utils, web3.eth) report metrics to CSV files, which we organize and store in a PostgreSQL relational database to facilitate the analysis of the data regarding each experimental campaign. Currently, we run a set of predefined SQL queries to generate a report with the metrics collected for all runs.

IV. Case Study

This section presents a case study designed to show the usefulness of the benchmark by applying it in different settings. The primary goal is to show that the benchmark can effectively measure the performance of various types of applications, including smart contracts, which are expected to underperform in specific metric categories running in a blockchain environment. To this end, we evaluate three groups of contracts. Each group contains four smart contracts: one baseline contract (referred to as V0), with no known vulnerabilities, and three variants (referred to as V1, V2, and V3), each with a single artificially injected vulnerability. Using our benchmark, we collect metric values for all contracts and perform a pairwise comparison between the vulnerable and non-vulnerable versions within each group. This approach allows us to evaluate the effectiveness of our benchmark in signaling performance differences among these contract variations. The approach is illustrated in Figure 4. Detailed results, as well as all experimental data necessary to replicate the experiments, are available at [12].

We first selected three blockchain applications of different types belonging to distinct domains to form each of the three groups. In particular, a *financially-sensitive* application *RentRoom* [41] (referred as ROOM-V0), *ClassAttendance* [42] which is an application for which *storage is critical* (referred as CLASS-V0), and a *privacy-sensitive* application *EHRBlockchain* [43] (referred as EHR-V0). RentRoom facilitates room reservations by tracking availability, costs, and services, requiring payment via Ether to release a certain room. Attendance is designed to manage student-teacher meetings, enabling scheduling, recording meeting minutes on the blockchain, and listing available users for future sessions. EHRBlockchain securely stores patient health data using smart contracts, enabling doctors, patients, and auditors to manage medical records with specific permissions. This approach also

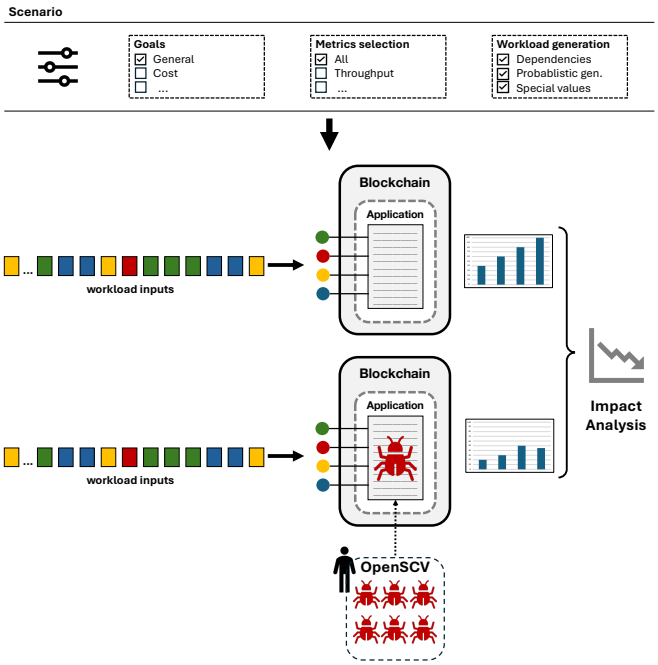

Fig. 4. Approach for assessing the impact of running vulnerable contracts on a blockchain.

gives patients control over their data, including the ability to delete sensitive information. Table II characterizes the three blockchain applications used as the baseline.

The applications presented in Table II were modified with the goal of having additional (and realistic) complexity. Regarding the RentRoom hotel application, we have included functionalities to i) extend the stay of a reservation (increase the number of days), ii) check the current guest bill, and iii) redeem discounts. Regarding the Class Attendance App, we have included a feature to record meeting minutes between students and teachers. In the EHRBlockchain application, we added the possibility of registering new patients.

In addition to the three non-vulnerable versions of the applications, we created nine additional versions, each containing a specific vulnerability. For each application, we selected vulnerabilities based on the nature of the application, prioritizing those that are likely to occur in that context and have the potential to impact the corresponding business logic. Table III describes the entire set of applications used, including details of the vulnerable versions, namely the selected vulnerabilities extracted from openSCV [44], the injection location, and the potential expected impact on the application.

We ran our benchmark against all 12 application versions. We set no particular importance on specific groups of metrics. The goal was to demonstrate that, first, the benchmark can be applied to various types of applications (i.e., with different interfaces, behaviors, and workload requirements); second, even in a neutral setting, performance differences in certain groups of metrics are noticeable and highlighted by the benchmark.

We configured each experimental run as follows. The first two minutes are used for ramp-up and are not included

TABLE II
DESCRIPTION OF THE THREE DECENTRALIZED APPLICATIONS USED AS A BASELINE IN OUR CASE STUDY.

dApp	ID	Operation	Return type	Modifier	Pay-able	Call dependency	Description
CLASS-V0 Class Attendance Management System	1	setEnroll (uint _roll, uint _year)	void	Owner		-	Starts a new academic year
	2	createStudent (uint _studId, uint _age,string memory _fName, string memory _lName, address _aStud)	void	-		-	The administrator creates student profiles and enrolls them in the system
	3	createTeacher (uint _teachId,string memory _fName, string memory _lName, string memory _discipline, address _aTeach)	void	-		-	The administrator creates teacher profiles and enrolls them in the system
	4	incrementAttendance (address _aTeach, address _aStud)	void	Teacher		2,3	Registers attendance with the participation of both teacher and student
	5	getStudents ()	object list	Teacher		2	Attempts to retrieve all students from the specified academic year
	6	getParticularStudent ()	object	Student		2	Tries to retrieve a single student record from a given academic year
	7	getTeacherList ()	object list	Teacher		3	Attempts to retrieve all teachers from the specified academic year
	8	addHistory (address _aStud, address _aTeach, string memory _comment)	void	Teacher		4	Records information about the teacher's assistance to the student.
EHR-V0 Electronic Health Record Blockchain	1	setInfo (string firstName, string lastName, string IID, string bdate, string email, string phone, string zip, string city, string encryption_key)	void	Owner		-	Creates a new Patient in the system
	2	start_visit (address _unique_id,uint _time)	string	Owner		1	Starts a new appointment for a patient
	3	addDoctors (address _doctor_address)	string	Owner		-	Add a Physician available to attend
	4	addAudits (address _audit_address)	string	Owner		-	Add Auditors available to attend
	5	doctor_print_record (address _unique_id)	array	Doctor		2	Physician prints the Patient record before starting the appointment
	6	doctor_query_record (address _unique_id)	array	Doctor		2	The physician retrieves the patient's records.
	7	doctor_update_record (address _unique_id)	array	Doctor		2	Physician updates a Patient's record based on the data of an appointment
	8	doctor_delete_record (address _unique_id)	array	Doctor		2	The Physician removes information previously recorded for a Patient
	9	get_record_details (address _unique_id)	string	Patient		2	The Patient queries the records pertaining to their own data
ROOM-V0 Room Renting	1	setReserveRoom ()	void	-	Y	-	Attempts to reserve a room, should it be available.
	2	setAddDaysToPay (uint256 _amount, uint8 _qtdDay)	void	-		1	Add charges, including daily rates and additional services, to the stay
	3	getCurrentBill ()	int	-		1	Returns the current bill for a single guest.
	4	getCurrentDay ()	int	-		2	Returns the number of days the guest has been staying
	5	getDiscount ()	int	-		2	Returns whether the guest received a discount during their stay
	6	setReleaseRoom ()	void	-	Y	1	Calculates and processes the payment required to check out of the room

TABLE III
DESCRIPTION OF THE VULNERABLE VERSIONS OF THE BASELINE APPLICATIONS.

Name	Target Operation	Injected Vulnerability	Expected Impact	Description
CLASS-V1	addHistory	5.16 Wrong Logic	Storage	Represents a programming mistake in the history registration function, in which an array of students is used for validation instead of an array of supervisors. It allows students to self-register their history without any consent from their supervisors.
CLASS-V2	createStudent	5.4.2 Wrong Selection of Guard Function	Cost	Wrong selection of guard function is, in our case, using 'assert' instead of 'require'. 'Assert' reverts a transaction but does not return the gas used. Thus, a version using 'assert' it is expected to have higher execution costs.
CLASS-V3	getTeacherList	8.2.1 Expose Private Data	Network	Vulnerability 8.2.1, in this context, allows both students and teachers to access unauthorized data (grades, comments), which likely impacts network metrics due to more traffic being generated.
EHR-V1	printRecord	5.7.2 No effect code execution	Cost	Additional code, with no effect on the contract state or the ledger, was added to the contract.
EHR-V2	printMyRecord	8.1.2 Owner Manipulation	Network	Vulnerability 8.1.2 allows unauthorized people (i.e., non-doctors) to access/manipulate patient records. Should generate additional network trafic.
EHR-V3	createPatientID	5.13.3 Read from Arbitrary Storage Location	Unknown	This operation includes a wrong access to an array, which results in a wrong patient identifier and ending up with the doctor creating an appointment for the wrong patient.
ROOM-V1	releaseRoom	5.4.2 Wrong Selection of Guard Function	Cost	Simulates the incorrect selection of a guard function (i.e., using 'assert' instead of 'require'). When verifying conditions for releasing a room, such as payment and occupancy checks, the failure of these conditions does not trigger a gas revert.
ROOM-V2	toString	5.7.2 No effect code execution	Cost	Additional code, with no effect on the contract state or the ledger, was added to the contract.
ROOM-V3	addDaysToPay	7.1.2 Integer Overflow	Cost	If a user adds sufficient days to its reservation it will overflow the variable storing the value and the user will be able to release the room without paying for it.

in the reporting. The middle 15 minutes correspond to the measurement interval, and the last two minutes are used for ramp-down and are also not considered in the benchmark report. The state was reset after each run to prevent the effect of previous runs from affecting the results (e.g., gas limits, gas prices, insufficient balances). For each new run, the applications were always executed starting from the genesis block (i.e., block 0), and all accounts were initialized with the same balance (i.e., 100 ether).

We added all necessary rules to the workload generator (e.g., call dependencies and special values required for certain operations) to the point where the generated workload could reasonably cover each contract. That is, all available operations should be exercised, and all useful code paths should be covered. Exception handling code, however, is not expected to be triggered by the workload, considering the nature of our three applications. The experiments were conducted on a single machine running Ubuntu 22.04.2 LTS, with 4 CPU cores (1 core per socket) and 8 GB of RAM. We also empirically determined the peak load our hardware could support and accordingly configured the benchmark to emulate 10 simultaneous clients that interact with the system, each

generating a constant load of 400 transactions per second. Throughout the whole experimental process, we monitored the machine resources to ensure there was no bottleneck (as it could influence the results).

V. Results and Discussion

This section presents and discusses the results obtained from executing our benchmark for each of the three groups of applications. We show results for all three groups in tables. These tables include the values of all metrics reported by bBench, as well as a column (RD) that shows the relative change of each metric compared to the reference value collected in V0. To simplify, we only show values that differ positively or negatively by more than 10% compared to the reference value.. We follow the lines of benchmarks, such as those offered by TPC or SPEC, and do not provide statistical testing. However, a user may use the benchmark's output to perform such tests.

A. Class Attendance Application

Table IV presents the benchmark results for the Class Attendance application. The following paragraphs go through each group of metrics and discuss cases of interest.

1) Network: As we can see, CLASS-V3 (with 8.2.1 Exposed Private Data vulnerability) stands out with the highest throughput, lowest latency, and also the highest committed transaction rate. The reason is that the vulnerability made public certain functions that handled private data of students and teachers (e.g., creating student profiles and scheduling appointments). These functions are computationally light. As a result, the workload associated with this contract included a greater number of function calls that could be executed faster compared to the other versions.

2) Resources: CLASS-V1 (with 5.16 Faulty Logic vulnerability) showed lower memory consumption when compared to the other versions. This is due to the fact that the implemented vulnerability caused the instruction flow to pass through a require statement more times without being reverted. This results in less memory consumption as the reversion process carried out by the infrastructure consumes more memory than the case where it is not carried out. This outcome may seem to be counter-intuitive, but this type of effect occurs in many cases (it is very context-specific), which end up positively affecting a particular metric but negatively affecting either other metrics or the business itself (as in this case).

3) Storage: CLASS-V1 (with 5.16 Faulty Logic vulnerability) also resulted in a smaller amount of data being stored in the smart contract's state variables (i.e., state size) because the instructions within the addHistory operation that use volatile storage were executed less frequently. Regarding block space consumption in the ledger (i.e., block size), we observed that all vulnerable contracts required more storage than CLASS-V0. Of all vulnerable versions, CLASS-V1 resulted in the generation of the largest number of records due to the creation of more academic history entries. While the injected vulnerability in CLASS-V1 had a relatively minor impact on volatile memory usage, it significantly contributed to the production of persistent data (i.e., due to the creation of more academic history entries), thus increasing the overall size of the stored information on the ledger.

4) Cost: CLASS-V3 (with 8.2.1 Exposed Private Data vulnerability) has a significantly higher execution cost. This occurred because the vulnerability activation led to additional data being exposed when compared to CLASS-V0, leading to increased Ether consumption.

B. Electronic Health Record Application

Table V presents the benchmark results for the Electronic Health Record application. The next paragraphs go through each group of metrics and discuss the respective cases of interest.

1) Network and Resources: As shown in Table V, both *network metrics* and *resource metrics* are barely affected by the activation of the various injected vulnerabilities.

2) Storage: The impact on storage was most evident in EHR-V3 (with 5.13.3 Read from Arbitrary Storage vulnerability), which contained a vulnerability allowing unrestricted access to array indices and resulted in less in-memory storage being used (smaller state size). The activation of the vulnerability led to numerous invalid references (e.g., to non-existent patient IDs) and caused several transactions to fail when creating appointments due to invalid patient IDs. These failed appointments were expected to be stored in memory, but this did not occur. As a result, the state variables in the affected contract showed a reduction in the size of stored data when compared to EHR-V0. On the other hand, in EHR-V2 (with 8.1.2 Owner Manipulation), the block size increased due to the execution of extra (unauthorized) function calls (e.g., a doctor retrieving information about patients he should not access).

3) Cost: In EHR-V1 (with 5.7.2 No effect code execution vulnerability), the larger contract size resulted in higher execution costs, as more gas is required to deploy larger contracts on the blockchain (the same happens with the execution of the additional code). The additional code was not large/complex enough to result in observable latency differences, and, given this scenario, the observed impact is mostly reflected in operational costs. We also see that EHR-V3 (with 5.13.3 Read from Arbitrary Storage Location vulnerability) resulted in lower Ether consumption (execution cost) compared to the baseline contract. This is explained by the fact that attempts to access nonexistent indices in state variables trigger revert operations, causing the execution to halt prematurely and thereby reducing gas usage when compared to EHR-V0.

C. Room Renting Application

Table VI presents the benchmark results for the Room Renting application. The next paragraphs discuss cases of interest in each group of metrics.

1) Network: We must begin by mentioning that the low rate of committed transactions in ROOM-V0 (10%) is due to the fact that reservations that cannot be concluded (i.e., due to lack of room availability) are correctly reverted. In practice, the code checks if the value of an integer variable is

TABLE IV
BENCHMARK RESULTS FOR THE CLASS ATTENDANCE APPLICATION.

dApp	Network								Resources				Storage				Cost			
	Throughput (Tx/s)	RD	Latency (ms)	RD	Commited Transactions (%)	RD	Commited Consensus (%)	RD	CPU(%)	RD	Mem(MB)	RD	State Size (Bytes)	RD	Block Size (Byte)	RD	Exc. Cost (ETH)	RD	Gas Price (ETH)	RD
CLASS-V0	344,15	–	29,29	–	70%	–	100%	–	82,54	–	1172,81	–	16800	–	349957	–	1,91	–	1E-09	–
CLASS-V1	342,90	–	28,18	–	61%	-0,14	100%	–	82,56	–	947,59	-0,19	13200	-0,21	489863	0,40	2,18	0,14	1E-09	–
CLASS-V2	351,03	–	25,71	-0,12	64%	–	100%	–	82,42	–	1154,83	–	16800	–	443010	0,27	2,36	0,24	1E-09	–
CLASS-V3	355,06	–	20,71	-0,29	80%	0,14	100%	–	81,71	–	1243,34	–	16800	–	469579	0,34	3,10	0,62	1E-09	–

TABLE V
BENCHMARK RESULTS FOR THE ELECTRONIC HEALTH RECORD APPLICATION.

dApp	Network								Resources				Storage				Cost			
	Throughput (Tx/s)	RD	Latency (ms)	RD	Commited Transactions (%)	RD	Commited Consensus (%)	RD	CPU(%)	RD	Mem(MB)	RD	State Size (Bytes)	RD	Block Size (Byte)	RD	Exc. Cost (ETH)	RD	Gas Price (ETH)	RD
EHR-V0	370,79	–	30,00	–	77%	–	100%	–	78,05	–	5980,95	–	24200	–	1855292	–	9,46	–	1E-09	–
EHR-V1	369,22	–	30,00	–	73%	–	100%	–	74,76	–	5787,65	–	26400	–	1894894	–	10,47	0,11	1E-09	–
EHR-V2	367,94	–	30,00	–	80%	–	100%	–	75,27	–	5536,19	–	24200	–	2074288	0,12	11,05	0,17	1E-09	–
EHR-V3	369,03	–	30,00	–	80%	–	100%	–	76,78	–	5449,38	–	19800	-0,18	1904775	–	8,32	-0,12	1E-09	–

TABLE VI
BENCHMARK RESULTS FOR THE ROOM RENTING APPLICATION.

dApp	Network								Resources				Storage				Cost			
	Throughput (Tx/s)	RD	Latency (ms)	RD	Commited Transactions (%)	RD	Commited Consensus (%)	RD	CPU(%)	RD	Mem(MB)	RD	State Size (Bytes)	RD	Block Size (Byte)	RD	Exc. Cost (ETH)	RD	Gas Price (ETH)	RD
ROOM-V0	334,56	–	22,73	–	10,3%	–	100%	–	77,56	–	4589,80	–	2200	–	2151103	–	20,08	–	1E-09	–
ROOM-V1	330,23	–	27,00	0,19	10,1%	–	100%	–	75,60	–	4361,85	–	2000	–	2204843	–	25,54	0,27	1E-09	–
ROOM-V2	355,70	–	24,00	–	8,1%	-0,22	100%	–	77,15	–	4595,30	–	2000	–	2748311	0,28	24,42	0,22	1E-09	–
ROOM-V3	369,90	0,11	24,00	–	95,8%	8,26	100%	–	75,71	–	4577,60	–	2000	–	2700086	0,26	28,17	0,40	1E-09	–

about to exceed the maximum value supported by its data type and reverts the transaction if so. Considering this, we observe ROOM-V3 (with 7.1.2 Integer Overflow vulnerability) with a high number of committed transactions (96%). The reason is that in ROOM-V3, the exception is not raised, and therefore, the transaction is not reverted, resulting in the generation of multiple invalid transactions with invalid data.

We also see that ROOM-V3 has a higher throughput, which is expected due to the high number of committed transactions. However, this gain in throughput is lower than we could expect. The reason is that in ROOM-V0, most transactions are reverted early more often, i.e., when an overflow would occur. These early reverts consume less gas and processing time compared to fully committed transactions. In ROOM-V3, although nearly all transactions are committed, they involve heavier state changes, often producing invalid but fully executed transactions. These transactions consume more gas and computation, which limits the number of transactions the system can process per unit of time. In practice, the overall system is slower in ROOM-V3 due to the higher cost of fully processing invalid transactions, resulting in an increase in throughput that is not proportional to the increase in committed transactions.

Regarding latency, it is worth mentioning that ROOM-V1 (with 5.4.2 Wrong Selection of Guard Function vulnerability) is the most affected. In this case, *require* statements were replaced with *assert* statements in a frequently called function. The assert statements take longer to execute, which is the reason for the higher latency observed in this version.

2) Resources: The performance variations observed in the resources group are all under 5%. The activation of the injected vulnerability led to no substantial change in this group of metrics.

3) Storage: The results for ROOM-V2 (with 5.7.2 No Effect Code Execution vulnerability) and ROOM-V3 show an increase in block size. The presence of no-effect extraneous code in ROOM-V2 increased gas consumption and contributed to the growth of the ledger data. As for ROOM-V3, the absence of an exception that would protect an integer overflow allows transactions to proceed incorrectly, which results in a higher number of faulty records.

4) Cost: ROOM-V1 resulted in higher execution cost due to the fact that, besides correct transactions, additional incorrect transactions are successfully executed and recorded on the ledger when this type of bug is present. The same happened with ROOM-V3, in which we have an overflow being triggered multiple times, resulting in more code being executed, which demands more gas. In ROOM-V2, the additional no-effect code simply increased the execution cost.

D. Additional Remarks

Some metrics, such as gas price, were not useful for comparison purposes in our experiments. This occurred as network price fluctuations were observed only during the ramp-up period. At the beginning of the execution, costs peaked at 1.7×10^{-9} and later stabilized at 1.0×10^{-9}. This behavior is attributed to the higher costs incurred during contract deployment and initial calls. After reaching the benchmark measurement period, gas prices associated with contract

execution remained stable. Since the contracts were executed on a local network without an active distributed consensus mechanism, Hardhat did not increase gas prices to regulate network usage.

The committed consensus was another metric with no variability, i.e., all runs reached 100%, meaning that the number of committed transactions matched exactly the number of transactions validated by the consensus mechanism. This outcome is due to the use of a local network environment, where the distributed and different behavior of miners is not simulated. In this setup, each block is generated instantaneously by a single miner, eliminating competition between branches of the blockchain and ensuring that valid blocks are never discarded. This is also something that we may evaluate in future work.

In addition to the goal of running the benchmark to understand possible differences within each of the three groups of applications, it also became obvious that different general performance characteristics exist across the groups. EHR-V0 demonstrated the best network and storage performance, while CLASS-V0 utilized significantly fewer resources than the remaining ones and incurred the lowest execution cost.

VI. Threats to Validity and Limitations

In this section, we discuss the possible **threats to the validity** of this study and corresponding mitigation actions, as well as the **limitations** of the work.

Construct validity is related to whether we are accurately capturing the performance characteristics we claim to measure. There is a risk that the selected metrics may not fully represent the system's performance under test, or that we may have missed some other relevant metrics. We tried to mitigate this threat by identifying metrics based on the state of the art and extending them to cover typical blockchain-specific concerns (e.g., operational costs, block size, storage usage).

Internal validity is concerned with unknown influences that may affect the independent and dependent variables. Given the complexity of the entire setup, aspects such as the network emulation environment or unknown interactions between the tools that comprise our toolset may introduce noise or have an unknown effect on the results. Also, given the randomness of the workload, we end up measuring the effect of a mix of transactions (some activate the vulnerability, while others do not). To mitigate these aspects, we repeated the experiments 3 times. We also reset the state after each experimental run, ensuring that the effects of a previous run do not accumulate in the next one.

Conclusion validity affects the ability to draw correct conclusions. The benchmark was capable of signaling performance differences originating from different types of applications and the injected vulnerabilities, but to allow for statistical analysis to take place, we would need additional experimental data. Still, to mitigate this threat, we did repeat the experiments three times to find little variability in the results.

External validity is related to the ability to generalize our benchmark results to other platforms or application domains. Currently, our benchmark is limited to Ethereum-based applications written in Solidity. Although we still do not support other blockchain environments (e.g., Hyperledger Fabric, Solana), we tried to mitigate this threat by successfully assessing different applications with different requirements and belonging to three different domains.

Our benchmark has the following **limitations**. Caliper does not allow the generation or injection of random parameters in the constructor of smart contracts. To allow our benchmark to execute, we modified the contracts by removing the constructor parameters and introducing an additional public initialization function, which is invoked shortly after the emulation begins.

Fallback functions are triggered when a contract receives a call to a function that does not exist, either because the function name is incorrect or the provided parameters do not match any of the defined function signatures [44]. This behavior cannot be currently simulated in our benchmark, as Caliper's underlying framework (i.e., Ethereum-connect.js) does not allow the execution of undefined functions or valid functions with incorrect parameters. Instead, such calls result in an exception that stops execution. This restriction is understandable as the tool must enforce basic input validation to avoid unintended behavior, but it also prevents testing fallback functionality.

VII. Conclusion

This work presented a new performance benchmark specifically designed for blockchain applications to measure network behavior, storage usage, computational resources, and operational cost. We built our solution based on established benchmarking principles and on the improvement of existing tools, such as Hyperledger Caliper, to retrieve additional metrics. We selected three applications of different types and domains, each with three additional vulnerable versions. bBench was able to quantify expected performance differences and also signal cases of interest in the different experimental scenarios.

In future work, we intend to further simplify the usage of the benchmark and extend it to be usable on various platforms, supporting additional smart contract languages. There are other aspects we intend to pursue, such as capturing real behaviors from a blockchain network like Ethereum and integrating aspects like gas usage distributions (dynamic adjustments to the gas price), or access patterns for more realistic workloads, or even network impairments (e.g., bandwidth limitations, packet drops).

Acknowledgments

This work is partially financed through national funds by FCT - Fundação para a Ciência e a Tecnologia, I.P., in the framework of the Project UIDB/00326/2025 and UIDP/00326/2025; by Project "Agenda Mobilizadora Sines Nexus". ref. No. 7113, supported by the Recovery and Resilience Plan (PRR) and by the European Funds Next Generation EU, following Notice No. 02/C05-i01/2022, Component 5 - Capitalization and Business Innovation - Mobilizing Agendas

for Business Innovation; and by Foundation for Science and Technology (FCT), Grant Nr. 2023.03131.BD.

REFERENCES

[1] V. Gramoli, R. Guerraoui, A. Lebedev, C. Natoli, and G. Voron, "Diablo: A Benchmark Suite for Blockchains," in *Proceedings of the Eighteenth European Conference on Computer Systems*. New York, NY, USA: ACM, may 2023, pp. 540–556.

[2] C. Yue, M. Zhang, C. Zhu, G. Chen, D. Loghin, and B. C. Ooi, "VeriBench: Analyzing the Performance of Database Systems with Verifiability," *Proceedings of the VLDB Endowment*, vol. 16, no. 9, pp. 2145–2157, may 2023.

[3] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008.

[4] J. Gray, *Benchmark handbook: for database and transaction processing systems*. Morgan Kaufmann Publishers Inc., 1992.

[5] K. Wolter, A. Avritzer, M. Vieira, and A. v. Moorsel, *Resilience Assessment and Evaluation of Computing Systems*. Springer Publishing Company, Incorporated, 2012.

[6] M. Vieira and H. Madeira, "A dependability benchmark for oltp application environments," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. VLDB Endowment, 2003, p. 742–753.

[7] M. Poess, "New initiatives in the tpc," in *Performance Evaluation and Benchmarking*, R. Nambiar and M. Poess, Eds. Cham: Springer Nature Switzerland, 2023, pp. 127–148.

[8] H. Brunst, S. Chandrasekaran, F. M. Ciorba, N. Hagerty, R. Henschel, G. Juckeland, J. Li, V. G. M. Vergara, S. Wienke, and M. Zavala, "First experiences in performance benchmarking with the new spechpc 2021 suites," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 675–684.

[9] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, may 2017, pp. 1085–1100.

[10] H. Pan, X. Duan, Y. Wu, L. Tseng, M. Aloqaily, and A. Boukerche, "BBB: A Lightweight Approach to Evaluate Private Blockchains in Clouds," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. New York, NY, USA: IEEE, dec 2020, pp. 1–6.

[11] M. Rasolroveicy, W. Haouari, and M. Fokaefs, "Blockcompass: A benchmarking platform for blockchain performance," *IEEE Transactions on Computers*, vol. 73, no. 8, pp. 2111–2122, 2024.

[12] Anonymous, "bbench: A comprehensive benchmark for blockchain applications – supplementary material," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15379337

[13] Standard Performance Evaluation Corporation, "SPEC Benchmarks and Performance Standards," https://www.spec.org, 2025, accessed: 2025-05-12.

[14] Transaction Processing Performance Council, "TPC Benchmark Standards and Results," https://www.tpc.org, 2025, accessed: 2025-05-12.

[15] Mininet Developers, "An instant virtual network on your laptop (or other pc)," 2024. [Online]. Available: https://mininet.org/

[16] D. Saingre, T. Ledoux, and J.-M. Menaud, "BCTMark: a Framework for Benchmarking Blockchain Technologies," in *IEEE/ACS 17th International Conf. on Computer Systems and Applications (AICCSA)*, 2020.

[17] B. Nasrulin, M. De Vos, G. Ishmaev, and J. Pouwelse, "Gromit: Benchmarking the Performance and Scalability of Blockchain Systems," in *IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, 2022, pp. 56–63.

[18] B. Bhushan, C. Sahoo, P. Sinha, and A. Khamparia, "Unification of blockchain and internet of things (biot): requirements, working model, challenges and future directions," *Wireless Networks*, vol. 27, 2021.

[19] N. Barman, G. Deepak, and M. G. Martini, "Blockchain for video streaming: Opportunities, challenges, and open issues," *Computer*, vol. 53, no. 7, pp. 45–56, 2020.

[20] M. N. Uddin, A. H. M. A. Hasnat, S. Nasrin, M. S. Alam, and M. A. Yousuf, "Secure file sharing system using blockchain, ipfs and pki technologies," in *2021 5th International Conference on Electrical Information and Communication Technology (EICT)*, 2021, pp. 1–5.

[21] M. Attaran and A. Gunasekaran, *Blockchain for Gaming*. Cham: Springer International Publishing, 2019, pp. 85–88.

[22] M. Bartoletti and L. Pompianu, *An Analysis of Bitcoin OP_RETURN Metadata*, 2017, pp. 218–230.

[23] L. Wang, Z. Guan, Z. Chen, and M. Hu, "schain: An efficient and secure solution for improving blockchain storage," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 3662–3676, 2023.

[24] C. Wu, M. J. Amiri, J. Asch, H. Nagda, Q. Zhang, and B. T. Loo, "Flexchain: an elastic disaggregated blockchain," *Proc. VLDB Endow.*, vol. 16, no. 1, p. 23–36, Sep. 2022.

[25] C. Melo, F. Oliveira, J. Dantas, J. Araujo, P. Pereira, R. Maciel, and P. Maciel, "Performance and availability evaluation of the blockchain platform hyperledger fabric," *The Journal of Supercomputing*, vol. 78, pp. 12 505–12 527, 7 2022.

[26] Z. Feng, Y. Li, and X. Ma, "Blockchain-oriented approach for detecting cyber-attack transactions," *Financial Innovation*, vol. 9, p. 81, 5 2023.

[27] F. R. Vidal, N. Ivaki, and N. Laranjeiro, "Vulnerability detection techniques for smart contracts: A systematic literature review," *Journal of Systems and Software*, p. 112160, 2024.

[28] C. Ma, W. Song, and J. Huang, "Transracer: Function dependence-guided transaction race detection for smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. ACM, 2023, p. 947–959.

[29] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering." ACM, 2014, pp. 1–10.

[30] A. Klenik and I. Kocsis, "Porting a benchmark with a classic workload to blockchain," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. New York, NY, USA: ACM, apr 2022, pp. 290–298.

[31] M. Touloupou, K. Christodoulou, A. Inglezakis, E. Iosif, and M. Themistocleous, "Benchmarking blockchains: The case of xrp ledger and beyond," 2022. [Online]. Available: http://hdl.handle.net/10125/80070

[32] G. Bubkoo, "Library for returning a random email address." 2024. [Online]. Available: https://www.npmjs.com/package/random-email

[33] ——, "Generate a random (u.s.) zip code," 2024. [Online]. Available: https://www.npmjs.com/package/random-zipcode

[34] ——, "Generate a random (u.s.) phone number," 2024. [Online]. Available: https://github.com/RoyLeibo/phone-number-generator-js

[35] S. Lee and J. S. Shin, "A new location verification protocol and blockchain-based drone rental mechanism in smart farming," *Computers and Electronics in Agriculture*, vol. 214, p. 108267, 2023.

[36] N. Foundation, "Understanding The DAO Attack," 2024. [Online]. Available: https://hardhat.org/

[37] H. Singh, "Chapter 6 - dapps: Decentralized applications for blockchains," in *Distributed Computing to Blockchain*, R. Pandey, S. Goundar, and S. Fatima, Eds., 2023.

[38] L. Palechor and C.-P. Bezemer, "How are solidity smart contracts tested in open source projects? an exploratory study," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. ACM, 2022, p. 165–169.

[39] J. Minango, M. Zambrano, and C. Minaya, "Exploring the use of blockchain for academic certificates: Development, testing, and deployment," in *Innovation and Research – Smart Technologies & Systems*, M. Z. Vizuete, M. Botto-Tobar, S. Casillas, C. Gonzalez, C. Sánchez, G. Gomes, and B. Durakovic, Eds. Springer, 2024, pp. 123–137.

[40] "Hyperledger Caliper." [Online]. Available: hyperledger.github.io/caliper/

[41] Samnang, "Solidity examples – hotel booking smart contract," https://github.com/samnang/solidity-examples/tree/main/contracts/sample_apps/03_hotel, 2023, accessed: 2025-05-12.

[42] Subha-Brata, "Student attendance management system using blockchain," https://github.com/Subha-Brata/studentAttendanceManagementSystem_Bockchain, 2023, accessed: 2025-05-12.

[43] Cumbul, "Ehrblockchain: Blockchain-based electronic health record system," https://github.com/cumbul/EHRBlockchain, 2023, accessed: 2025-05-12.

[44] F. R. Vidal, N. Ivaki, and N. Laranjeiro, "OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities," *Empirical Software Engineering*, vol. 29, no. 4, p. 101, jul 2024.