Using Fault Injection to Assess Blockchain Systems in Presence of Faulty Smart Contracts

ÁKOS HAJDU [ID]1, NAGHMEH IVAKI [ID]2, (Member, IEEE), IMRE KOCSIS1, ATTILA KLENIK1,
LÁSZLÓ GÖNCZY1, NUNO LARANJEIRO [ID]2, (Member, IEEE),
HENRIQUE MADEIRA2, (Member, IEEE), AND ANDRÁS PATARICZA1

1Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1117 Budapest, Hungary
2CISUC, Department of Informatics Engineering, University of Coimbra, 3030-290 Coimbra, Portugal

Corresponding author: Ákos Hajdu (hajdua@mit.bme.hu)

ABSTRACT Blockchain has become particularly popular due to its promise to support business-critical services in very different domains (e.g., retail, healthcare). Blockchain systems rely on complex middleware, like Ethereum or Hyperledger Fabric, that allow running smart contracts, which specify business logic in cooperative applications. The presence of software defects in these contracts has notably caused failures, including severe security problems. In this article, we use software-implemented fault injection (SWIFI) to assess the behavior of permissioned blockchain systems in the presence of faulty smart contracts. We emulate the occurrence of general software faults and also blockchain-specific software faults (e.g., missing *require* on transaction sender) in smart contracts code and observe the impact on the overall system dependability in terms of reliability and integrity. We also analyze the effectiveness of formal verification and runtime protection mechanisms in detecting the injected faults. Results indicate that formal verification and runtime protections have to complement built-in platform checks to guarantee proper dependability of blockchain systems. The work presented in this article allows smart contract developers to become aware of possible faults in smart contracts and to understand the impact of their presence. It also provides valuable information for middleware developers to improve the overall fault tolerance of their systems.

INDEX TERMS Blockchain systems, dependability, fault injection, formal verification, smart contracts.

I. INTRODUCTION

The blockchain is a software-based distributed ledger technology, which, just like in a hard copy ledger, is a way for cooperating partners to store and track transaction records [1]. Its growing popularity can be attributed to core properties such as decentralization, immutability, security, and transparency [2] and to the fact that it naturally applies to many contexts, from supply chain management through financial markets to healthcare [3]–[5].

In a blockchain environment, code supplied by some of the users of the system defines transaction logic that is

application-specific. The Ethereum technology [6] made such programs, similar to stored procedures and named *smart contracts* [7], popular as a general mechanism for executing client-proposed ledger transactions. Cryptographically linked (*chained*) *blocks* of the proposed transactions have to pass a group consensus among a set of nodes (*peers*), maintaining the distributed, shared ledger. The group consensus accounts for the order as well as any consequences of smart contract based transactions, which lead to modifications in the ledger state.

Blockchain systems are commonly viewed as being extremely secure and dependable, but the reality is much more nuanced. Regarding security, while consensus may probably be attack-resistant, it does not address the fact that

The associate editor coordinating the review of this manuscript and approving it for publication was Wenbing Zhao [ID].

deployed smart contract code may hold software defects, including vulnerabilities [8], [9], allowing for certain executions to harm the system. Regarding the general concept of dependability, the system's key properties like ledger integrity, service reliability, and service availability are a function of a complex interplay between different mechanisms that target different non-functional properties, which are challenging to check using traditional verification techniques.

Within such a complex environment, quality of smart contracts (i.e., the absence of software faults) plays a vital role. Residual bugs in smart contracts (software faults that may be activated due to absent or inadequate protection mechanisms, or that were not detected by verification activities like testing or static analysis) have been linked to failures in blockchain systems and are known to be the cause of severe security problems [9], [10], as, for instance, the infamous DAO attack demonstrated [11].

In this article, **we present a software-implemented fault injection (SWIFI) technique [12], [13] that has been tailored for evaluating the behavior of blockchain systems in the presence of faulty smart contracts**. We target smart contracts written in the popular Solidity language [14], which is being supported by an increasing number of blockchain platforms. Starting with a general software fault model for software fault injection as basis [15], we eliminate rare software faults and faults that do not apply to our context (e.g., faults that cannot affect programs written in Solidity) and complement the fault model with **smart contract-specific faults**, based on 515 faults listed at the NIST National Vulnerability Database [10] and blockchain and smart contract bugs presented in the literature. Based on the resulting fault model, we inject software faults into the smart contract abstract syntax tree (AST) to generate faulty smart contracts, which are then executed on a blockchain platform.

The blockchain system is observed and analyzed considering the following perspectives: i) *reliability (or correctness)*: divergence from correct behavior regarding the external behavior of the system (observable by the client); ii) *integrity evaluation*: the ledger state integrity violation (either observable by the client or not). We complement the empirical assessment using *formal verification*, i.e., static analysis for possible detection of injected faults and verification of incorrect functioning of smart contracts (to understand which faults and incorrect behavior would be captured before deployment and which would likely escape to production).

We carried out an experimental evaluation using *Hyperledger Fabric* [16], the market-leading business blockchain platform, and a set of 15 smart contracts (5 base versions, 5 versions modified to hold extensive protections against invalid inputs, and 5 versions with no protection) in which we injected faults, which resulted in a total of 651 faulty versions. Results show the distinct effects of the different types of injected faults and strongly suggest that using formal verification and additional code-level protection mechanisms can prevent the occurrence or activation of different types of

faults, thus being of utmost importance for developing and deploying critical blockchain applications.

The main contributions of this article are the following.

- A *smart contract software fault model* of 51 fault types (33 general and 18 smart contract specific);
- The definition of an *approach for injecting faults* into smart contracts, which is independent from the programming language used;
- A *fault injector tool* implemented for Solidity smart contracts;
- The definition of a set of experiments to show i) the impact of different faults on the behavior of a blockchain system in terms of reliability and integrity; and ii) the effectiveness and the complementary nature of several verification and protection mechanisms in place including *smart contract formal verification*, *runtime platform checks*, and *contract-level protections* under our fault model.

The paper is organized as follows. The next section, Section II, introduces the background and related work and Section III presents the smart contract fault model and the fault injection approach used. Section IV introduces the experimental setup and implementation, and the evaluation process followed. Section V presents and discusses the results obtained, and Section VI highlights the threats to the validity of the results and the methods used in this work. Finally, Section VII concludes the paper.

II. BACKGROUND AND RELATED WORK

In this section, we present the necessary background on blockchain technology and smart contracts and review existing works that address dependability (and security) in this environment.

A. BLOCKCHAIN TECHNOLOGIES

Blockchain is, in practice, a digital implementation of the hardcopy transaction ledger [17], which acts as a highly secure and resilient database and execution environment for the business logic deployed on it in the form of smart contracts. Nowadays, digital ledgers are used in businesses and organizations in a variety of applications, such as the movement of assets or properties, recording contracts, buy-sell deals, and liabilities documentation. A large number of increasingly mature blockchain platforms are available today, facilitating the creation of public access, ''unpermissioned'' consensus participation networks with cryptocurrencies, as well as the ones that can be accessed only by specific business consortia in a fully permissioned manner.

Ethereum is a key technology in both worlds. A core part of its specification is a simple bytecode virtual machine for smart contract execution, the *Ethereum Virtual Machine* (EVM) [6]. The EVM is a general-purpose, Turing-complete virtual machine that provides access to the ledger state largely via key-value style CRUD (create, read, update and delete) operations. As of today, Solidity is the most popular language targeting the EVM [14]. However, Solidity is also becoming
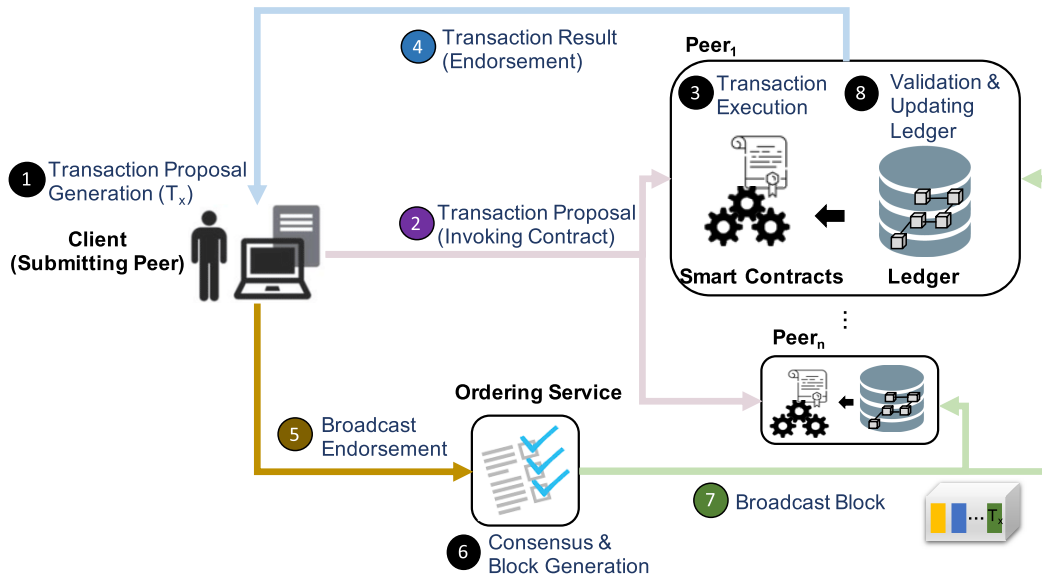
FIGURE 1. Transaction execution in Hyperledger Fabric.

available on blockchain platforms other than Ethereum. For instance, the Hyperledger Burrow [18] EVM implementation can be deployed in Hyperledger Fabric.

Solidity is a statically typed, contract-oriented language with high-level facilities such as complex data types, multiple inheritance, and libraries, with a syntax resembling ECMAScript. The Solidity programming model natively handles cryptocurrency (Ether) movements between pseudonymous (cryptographic key authenticated) parties, and EVM-based execution is defined in terms of the calling party offering up-front cryptocurrency payment as fuel (*gas*) for smart contract function call executions. If the offered gas runs out before termination, the call fails. Regardless, these peculiarities can be easily hidden or circumvented in business-oriented blockchain networks and applications. Our paper does not require a more in-depth introduction of either Ethereum or the EVM; the interested reader is kindly referred to the aforementioned references.

For our experiments, we chose Hyperledger Fabric [16], the platform supporting the creation of "bespoke", business consortium specific, permissioned blockchain systems to the most extent. In Hyperledger Fabric, a set of *organizations* as a *consortium* form a blockchain network from peers controlled by them. The network is directly accessible only to their systems and users.

Compared to most blockchain platforms, transaction execution and block consensus are fairly unique in Hyperledger Fabric. Fig. 1 presents an overview. The core idea is that a client asks peers of organizations to *endorse* a proposed transaction (essentially, smart contract function call) by executing it on their current ledger state and replying with the so-called *read-write set* upon success – instead of modifying their ledger state. The read-write set contains the (versioned) set of variables and the updates a successful execution would

perform. Given enough matching (signed) endorsements, the *ordering service* accepts the proposal and includes it in the next block, to be distributed to the peers for ledger state updates. The full protocol, which is described in detail in [16] and in the Hyperledger Fabric documentation, is summarized below and presented in Fig. 1.

1) A client generates and digitally signs a transaction proposal;
2) The client sends the transaction proposal directly to one or more so-called *endorsing* peers of one or more organizations participating in the blockchain;
3) After checking the identity of the client, the peers execute the transaction (smart contract function call) using their current ledger state – but do not update it; in essence, performing *transaction simulation*;
4) The peers respond to the client with a digitally signed *read-write set* of such ledger variables that the actual transaction execution would read and write;
5) The client collects the responses and submits them to an ordering service (provided by a disinterested third party, or a group consensus protocol across the organizations);
6) The ordering service performs a consensus protocol and forms blocks of the multiparty-endorsed transaction proposals;
7) The ordering service disseminates the blocks to the peers of the organizations;
8) Peers validate the transactions in the blocks and modify their ledger state.

B. RELATED WORK

In business or mission-critical environments, the presence of software bugs in a blockchain system is a problem that can bring severe consequences to the business or mission itself,

impairing its dependability [9]. This is aggravated by the fact that, in some cases, the programming languages used for writing the contracts are not mainstream languages (e.g., Solidity), and developers may lack the necessary expertise, leading to the deployment of contracts holding residual bugs.

Despite several attempts in the literature to define best practices for implementing secure and fault-free smart contracts [19], [20], developing contracts without faults seems to be very difficult, if possible. As smart contracts often manipulate (or provide accounting for) valuable assets, verification of smart contracts is playing an increasingly important role [17], but as the blockchain technology is still relatively recent, there are just a few, and not really effective, tools [9], [21], [22] (e.g., static analyzers [23]–[29], theorem provers [30]–[34], and SMT-based tools [35]–[38]).

Testing tools [39], [40] have been developed to detect both bugs and vulnerabilities, as well as formal approaches to assure the correctness of smart contracts [9], [21], [22].

Most of these approaches are adaptations of, or built on, existing methods, inheriting their advantages and limitations. For example, static and symbolic analyzers are effective at finding vulnerable patterns in smart contracts [23]–[29], but might yield a high rate of false alarms and miss bugs related to the business logic. Formal methods provide a sound basis, but usually require formal specification [33], [36], [38] or assisted proofs [30]–[34].

Our work is focused on using the software-implemented fault injection (SWIFI) technique [12], [13], which is a vital tool for evaluating the dependability of software systems, especially critical systems. Typical SWIFI approaches include the injection of code changes in certain components of the system (that emulate the most frequent types of bugs introduced during development) or the direct injection of the effects of faults (e.g., faulty data or user inputs). Regardless of the technique, the goal is to be able to evaluate how a faulty component may affect the behavior of the overall system, which also allows for understanding the effectiveness of the fault tolerance mechanisms in place [41].

We can find a few works in the literature, recently published under the title of smart contract *mutation testing* [42]–[44], that follow a similar approach (i.e., injection of faults into AST) for the generation of faulty smart contracts for testing purposes. However, these works are more focused on implementing the respective mutation testing tools rather than the assessment of the blockchain systems' behavior and the effectiveness of the protection and verification mechanisms in place. In addition, fuzzing tools [45]–[47] are related to our work in the sense they also generate inputs to check for vulnerabilities. However, existing fuzzers only consider the original (existing) contracts and do not include fault injection to evaluate the impact of faults in smart contracts.

Our approach of using software fault injection can be considered as a complementary technique to assess the effects of both common programming errors and smart contract-specific faults. Furthermore, software fault injection can also be used as a benchmark to evaluate the precision and recall of existing tools such as static analyzers and fuzzers.

III. FAULT INJECTION INTO SMART CONTRACTS

This section describes the overall approach for assessing the reliability of a blockchain system in the presence of faulty smart contracts. The approach relies on the execution of smart contract transactions accompanied by the formal verification of possible defects, which works as a complement to the runtime execution (to understand if certain defects could be caught before runtime). In the next subsections, we present: i) an overview of the approach; ii) the definition of the fault model used and its application to smart contracts; iii) the workload generation process; iv) the process of formal verification applied to contracts, and v) how the collected data is analyzed.

A. OVERVIEW OF THE APPROACH

Fig. 2 presents a general view of the fault injection environment with the components involved, and the process followed to inject faults into the system under assessment (SUA).

In general, to apply a fault injection technique, we need to address the following main questions: which parts of the system are targeted for fault injection (i.e., fault injection location), what faults to be injected, and when to inject the faults? In this work, the target of fault injection within the blockchain system is the **smart contract**. Thus, it is one of the inputs for the approach (depicted at the top left of Fig. 2). One of the challenges involved in the use of a fault injection technique is the definition of a **fault model** (at the top of Fig. 2) that should be composed of representative types of faults, which will be injected, in our case, into the smart contracts. The definition of the fault model essentially covers general cases of typical programming mistakes [15], [48] and specific cases of smart contract related faults [10].

To **inject faults into smart contracts**, we integrate changes (i.e., fault models similar to mutations but based on realistic fault assumptions) into the source code of the smart contracts. The implementation of these changes (i.e., faults) strongly depends on the programming language used to write the smart contract. To eliminate this dependency, our **fault injector** first generates an Abstract Syntax Tree (AST), which is an abstract representation of the syntactic structure of source code for a certain smart contract source code provided as input. We then inject individual faults into the AST, which results in several faulty ASTs, each with exactly one injected fault. Then the faulty ASTs are converted back to code (Solidity code, in our case), resulting in faulty versions of smart contracts.

Both the **fault-free and faulty smart contracts are then executed** individually on the blockchain system, in an isolated environment. Fault-free runs are used as a reference to evaluate the results of faulty runs. A challenge at this point is the generation of **workloads** that are representative and
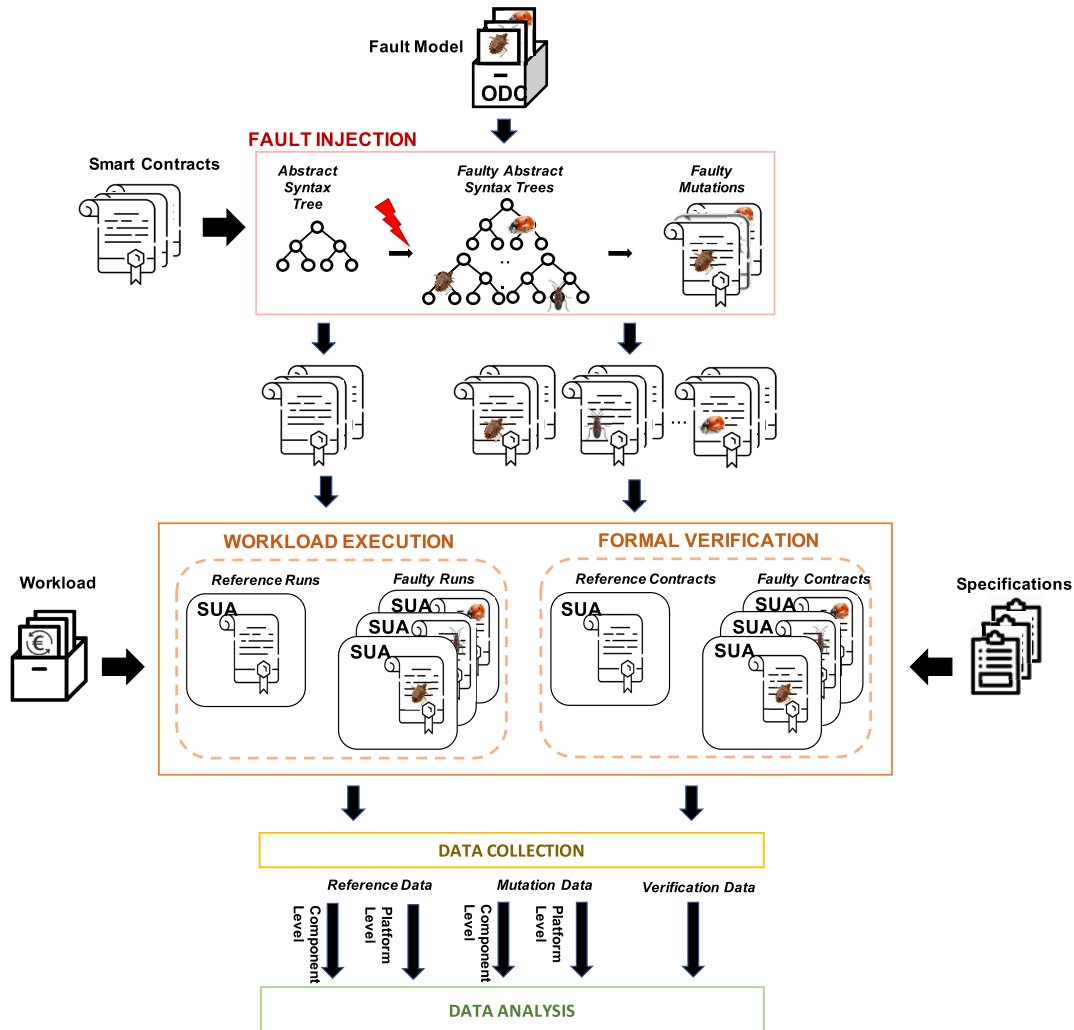
allow activating the artificially introduced faults during the execution of transactions.

We complement the assessment by performing **formal verification** over both reference contracts and the respective mutated contracts that are annotated with **specifications** (e.g., contract invariants, pre- and post-conditions) [36]. This allows us to assess the effects of *protection mechanisms* in smart contract code (e.g., integrity-protecting error detecting assertions and observability-increasing return statements) and to understand which faults and incorrect behavior would be captured before deployment. During assessment, the system under assessment (SUA) is monitored while executing the fault-free and faulty smart contracts, and all necessary **data are collected** and stored in log files for later analysis.

After finishing the runs, the **analysis of the data** is performed from the reliability (i.e., correct external and internal behavior) and integrity (i.e., ledger state integrity) perspectives at two levels: i) *component level*, which refers to the direct analysis of the behavior of the container running the

smart contracts, allowing us to understand the impact of the faults on the container in which the smart contract is being executed; ii) *platform level*, in which the analysis is performed at the level of the whole blockchain platform, i.e., to understand the impact of faults on the whole blockchain system setting. At each of these analysis levels, we have reference data corresponding to the outcome of fault-free runs (i.e., component level reference data and platform level reference data) and mutation data corresponding to the outcome of faulty smart contract runs (component level mutation data and platform level mutation data).

The details regarding the definition of the fault model, formal verification, and criteria used for behavior assessment are presented in the following subsections.

B. FAULT MODEL DEFINITION
Our fault model uses a combination of software faults covering two cases: i) general software faults [48] and ii) smart contract-specific faults, which include 515 faults listed at

TABLE 1. Smart contract specific faults reported in NVD [10].

Type of Vulnerability	Count	Percentage
Integer Overflow or Wraparound	474	92%
Improper Access Control	18	3,5%
Input Validation	9	1,7%
Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	8	1,6%
Use of Insufficiently Random Values	2	0,4%
Integer Underflow (Wrap or Wraparound)	2	0,4%
Integer Buffer Errors	1	0,2%
Out-of-bounds Read	1	0,2%

the NIST National Vulnerability Database [10] and also blockchain and smart contract bugs presented in the literature [24]–[26], [49], [50].

To build the fault model, we use the information provided by the Orthogonal Defect Classification (ODC) [48], which provides a useful foundation for emulating software faults. ODC is often used as a source of information to provide insights and feedback regarding the quality of software and the development process, but it also provides a useful foundation regarding the emulation of software faults. Building on ODC, an extensive classification of representative software faults was carried out in [15], in which the identification and classification of software faults are made from a fault injection perspective so that the identified faults can be easily emulated in code.

According to [15], the most representative types of defects, i.e., of software faults are (in order): i) Assignment (e.g., the value assigned to a variable is incorrect); ii) Checking (e.g., the input data is not validated correctly); iii) Interface (e.g., error in function call); iv) Algorithm (e.g., incorrect implementation of an algorithm); and v) Function (e.g., a functionality is affected). The nature of a defect type (i.e., the ODC qualifier [48]) can be one of the following cases: Missing construct (e.g., missing function call), Wrong construct (e.g., the wrong variable used in parameter of a function call), and Extraneous construct (e.g., extraneous function call).

We started the definition of our fault model based on the general results in [15], which, in total, identifies 62 software faults. From this list, we removed the ones that do not apply to our context, which uses a specific programming language (Solidity) and respective compiler. For instance, *Missing case: statement(s) inside a switch construct (MCS)* and *Wrong branch construct - goto instead break (WBCI)* are not applicable because there are no `goto` and `switch` statements in Solidity, or *Missing parameter in function call (MPFC)* and *wrong data types or conversion used (WSUT)* are also not applicable because they are caught by the compiler. After this analysis, 8 faults (out of 62) were excluded from the fault model.

We also excluded the less frequent faults (those appearing with a frequency lower than 1% (refer to [15] for more details about the frequency of the faults). *Wrong parenthesis in logical expression in parameters of function call (WPLP)*

is one example of these faults. This resulted in a total of 21 faults being excluded. For the time being, we have a list of 33 frequent generic software fault types, identified in the last column of Table 2 with the letter **G**, that apply to our field study.

In a second step, with the goal of increasing the representativeness of the fault model, we completed the list of faults with smart contract and Solidity specific faults. These faults are identified in the last column of Table 2 with the letter **S**. To define the specific faults, we collected and analyzed all smart contract faults reported in the National Vulnerability Database (NVD) [10] at the time of writing (a total of 515 vulnerabilities), as well as the blockchain and smart contract bugs presented and analyzed in the literature [24]–[26], [49], [50]. Table 1 summarizes all vulnerabilities reported in NVD, categorized into 8 distinct types and from which we focus on the more frequent ones.

As shown in Table 1, the most frequent vulnerability is *Integer Overflow or Wraparound* (92% of all reported vulnerabilities), which occurs when the result of an arithmetic computation exceeds the range of its type. When this occurs, the value may wrap to become a negative number or a very small number. For example, $127 + 1$ becomes -128, instead of 128, when one signed byte (8 bits) is used to store the result. Overflows do not cause a run-time error and can silently lead to incorrect calculations and undefined behavior. This becomes a severe issue when the result of an arithmetic computation is used to control loops, determine the size in operations such as memory allocation, or take some other action decision (e.g., used to control an authentication or authorization process). In some systems, it may even impact performance. Smart contract developers can avoid integer overflows using recognized integer handling libraries, such as *SafeMath* [51]. Thus, missing calls to these safe integer handling functions may lead to integer overflows. For this reason, fault 25.*a* (Missing calls to *SafeMath* (MCSM)) is defined under this category of faults (i.e., Integer Overflow or Wraparound) as a special case of the ODC fault `Missing function call (MFC)`.

The next frequent vulnerability, although not as frequent as the Integer Overflow or Wraparound, is *Improper Access Control* (3.5% of all reported vulnerabilities). This occurs when a smart contract does not restrict or incorrectly restricts access to a certain resource or function from an unauthorized

TABLE 2. Fault model.

ID	Fault Class	Fault Nature	Fault	Generic/Specific, Impl.
1	Assignment	Missing	variable initialization using a value (**MVIV**)	G , ✔
2			variable initialization using a expression (**MVIE**)	G , ✔
3			variable assignment using value (**MVAV**)	G , ✔
4			variable assignment using an expression (**MVAE**)	G , ✔
5			variable auto-increment (**MVAI**)	G , ✔
6			variable Auto-decrement (**MVAD**)	G , ✔
7		Wrong	arithmetic expression used in assignment (**WVAE**)	G , ✔
8			miss-by-one value used in variable initialization (**WVIM**)	G , ✔
9			value assigned to variable (**WVAV**)	G , ✘
10		Extraneous	variable assignment using another variable (**EVAV**)	G , ✘
11	Checking	Missing	"if" construct around statement (**MIA**)	G , ✔
11.a			*"require" on transaction sender (**MRTS**)*	S , ✔
11.b			*"require" on input variable(s) (**MRIV**)*	S , ✔
12			"OR EXPR" in expression used as branch condition (**MLOC**)	G , ✔
12.a			*"require" OR subexpression on transaction sender (**MROTS**)*	S , ✔
12.b			*"require" OR subexpression on input variable(s) (**MROIV**)*	S , ✔
12.c			*"if" const. OR subexpression on transaction sender (**MIOTS**)*	S , ✔
12.d			*"if" construct OR subexpression on input variable(s) (**MIOIV**)*	S , ✔
13			"AND EXPR" in expression used as branch condition (**MLAC**)	G , ✔
13.a			*"require" AND subexpression on transaction sender (**MRATS**)*	S , ✔
13.b			*"if" const. AND subexpression on transaction sender (**MIATS**)*	S , ✔
13.c			*"require" AND subexpression on input variable(s) (**MRAIV**)*	S , ✔
13.d			*"if" construct AND subexpression on input variable(s) (**MIAIV**)*	S , ✔
14		Wrong	parenthesis in logical expression used in assignment (**WPLC**)	G , ✘
15			logical expression used as branch condition (**WLEC**)	G , ✔
15.a			*logical expression in "require" over input variable(s) (**WRIV**)*	S , ✔
15.b			*logical expression in "if" const. over input variable(s) (**WIIV**)*	S , ✔
15.c			*logical expression in "require" over transaction sender (**WRTS**)*	S , ✔
15.d			*logical expression in "if" const. over transaction sender (**WITS**)*	S , ✔
16			arithmetic expression in branch condition (**WAEC**)	G , ✔
17	Interface	Missing	return statement (**MRS**)	G , ✔
18			"OR sub-expr" in parameters of function call (**MLOP**)	G , ✔
19			"AND sub-expr" in parameters of function call (**MLAP**)	G , ✔
20		Wrong	logical expression in parameters of function call (**WLEP**)	G , ✔
21			arithmetic expression in parameters of func. call (**WAEP**)	G , ✘
22			variable used in parameter of function call (**WPFV**)	G , ✘
23			return value (**WRV**)	G , ✘
24			*visibility (public) for private/internal function (**PVPF**)*	S , ✔
25	Algorithm	Missing	function call (**MFC**)	G , ✔
25.a			*call to SafeMath (**MCSM**)*	S , ✔
26			"if" construct plus statements (**MIFS**)	G , ✔
26.a			*"if" construct on transaction sender plus statements (**MITSS**)*	S , ✔
26.b			*"if" construct on input variable(s) plus statements (**MIIVS**)*	S , ✔
27			"if-else" construct plus statements (**MIES**)	G , ✔
28			"if" const. plus statements plus "else" before statements (**MIEB**)	G , ✘
29			iteration construct around statements (**MCA**)	G , ✔
30			small and localized part of the algorithm (**MLPA**)	G , ✘
31		Wrong	function called with same parameters (**WFCS**)	G , ✘
32			function called with different parameters (**WFCD**)	G , ✘
33			algorithm - code was misplaced (**WALR**)	G , ✔
34	Func.	Wrong	algorithm - large modifications (**WALL**)	G , ✘

category of faults (i.e., Improper Access Control) as special cases of the corresponding ODC faults.

Input Validation is the next frequent smart contract vulnerability (1.7% of all reported vulnerabilities), which refers to wrong or missing validation of inputs (input values are not validated correctly or are not validated at all), something that usually takes place at the beginning of transactions to revert the whole transaction when a condition does not hold. Wrong inputs may affect data flow and control flow of the transaction leading to various problems, such as robustness or security issues. The possible input validation checks in smart contracts are very diverse, depending on the type of input data and the business logic expressed by the contracts, but to emulate this kind of vulnerability, such validation checks must be eliminated or modified (e.g., by altering the checking conditions). Faults 11.*b* (Missing *require* on input variable(s) (MRIV)), 12.*b* (Missing *require* OR subexpression on input variable(s) (MROIV)), 12.*d* (Missing *if* construct OR subexpression on input variable(s) (MIOIV)), 13.*c* (Missing *require* AND subexpression on input variable(s) (MRAIV)), 13.*d* (Missing *if* construct AND subexpression on input variable(s) (MIAIV)), 15.*a* (Wrong logical expression in *require* over input variable(s) (WRIV)), 15.*b* (Wrong logical expression in *if* construct over input variable(s) (WIIV)), and 26.*b* Missing *if* construct on input variable(s) plus statements (MIIVS)) are defined under this category of vulnerabilities as special cases of the corresponding ODC faults.

Finally, we found *Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)* (1.6% of all reported vulnerabilities), which occurs when a non-cryptographic PRNG or a PRNG's algorithm that is not cryptographically strong is used in a context where security is important. For instance, if a non-cryptographic PRNG algorithm is used to generate a session ID (i.e., used for authentication and authorization), then an attacker might be able to easily guess the session ID and gain access to restricted resources. Although being an important security issue, since the impact of this vulnerability is very tightly coupled to the intention of the attacker and to the logic involved in the transactions and smart contracts, for the time being, we do not include it in our fault model.

As shown in the last column of Table 2, we implemented all smart contract specific faults and most (in total covering 78.4%) of the software generic faults (identified with ✔). We left a few of the generic faults unimplemented (the ones that appear in light grey in the table and identified with X), due to the fact that they do not seem to be very common and relevant in smart contracts according to the literature [24]–[26], [49], [50].

C. WORKLOAD GENERATION

Workload generation, in our context, is largely equivalent to automated test case generation. We are not aware of any such mature tool, so as part of our research, we defined a workload generation process that generates values for smart contract

user. Several faults in smart contracts may introduce such access control weaknesses, in particular, those that include incorrect or missing conditions required to be verified for the sender of the transactions or incorrect visibility defined for the resources. Faults 11.*a* (Missing *require* on transaction sender (MRTS)), 12.*a* (Missing *require* OR subexpression on transaction sender (MROTS)), 12.*c* (Missing *if* construct OR subexpression on transaction sender (MIOTS)), 13.*a* (Missing *require* AND subexpression on transaction sender (MRATS)), 13.*b* (Missing *if* construct AND subexpression on transaction sender (MIATS)), 15.*c* (Wrong logical expression in *require* over transaction sender (WRTS)), 15.*d* (Wrong logical expression in *if* construct over transaction sender (WITS)), 24 (Wrong visibility (public) for private/internal function (PVPF)), and 26.*a* (Missing *if* construct on transaction sender plus statements (MITSS)) are defined under this

function call input parameters based on their *type*, the *literals* appearing in the code, and *randomly*, as follows:

- *Type-based inputs*: Generate input values for each parameter based on its type. For Booleans, we generate both true and false, while for integers, we generate the minimum, the maximum, and zero. Input values for reference types, such as strings and arrays, are generated recursively for different lengths, including the length of zero.
- *Literal-based inputs*: Inputs are often compared to literals, which determines the control-flow inside the function. Therefore, we also extract each literal from the function and use them as inputs to the parameters with a matching type. Furthermore, for integer literals l we also include $l+1$ and $l-1$ to catch off-by-one errors.
- *Random inputs*: Generate a configurable number of random values for certain types. For integers, we generate random values within the minimum and maximum range. We also generate random strings and arrays by generating random elements of their sub-type and shuffling them randomly for different lengths.

D. FORMAL VERIFICATION

We complement the empirical assessment using *formal verification* (i.e., static code analysis) in order to understand how difficult it would be to detect injected faults and incorrect functioning of smart contracts. To do so, we execute a formal verification technique, namely *modular program verification* [52] over reference contracts and the respective mutated contracts. In modular program verification, the smart contract's source code should be annotated with *specifications* (e.g., contract invariants, pre- and postconditions) of the expected behavior. Each annotated contract is translated into a formal representation in an intermediate verification language [53], having mathematically precise semantics. Each function in this representation is then translated into a set of formulas (so-called verification conditions) that establish the correctness of the function.

The verifier discharges these formulas automatically using SMT (Satisfiability Modulo Theories) solvers [54] by checking their validity. The modular verification approach used in this work [36] targets the *functional correctness* of contracts, with respect to different kinds of failures, including expected and unexpected failures. An *expected failure* is a failure that is caused due to an explicit guard defined by developers (e.g., Solidity `require`, `revert` statements). An unexpected failure is any other failure (e.g., `assert`, overflow). A contract is reported as *correct* if there is no unexpected failure and all of its transactions that are finished without an expected failure, satisfy their specification.

E. BEHAVIOR ASSESSMENT

The last step of the approach is related to evaluating the behavior of the blockchain system in the presence of the injected faults, which is carried out in perspective with the fault-free runs. In this work, we concern about the faults' effects on the system's **reliability** (in *lato sensu*) and **integrity**. Notice that integrity errors may, or may not, affect the reliable behavior of the blockchain system.

Reliability evaluation mostly concerns divergence from correct behavior (i.e., regarding the external, observable, behavior). Evaluation of the *external behavior* is relatively simple in the case of smart contracts. We can check whether the expected outcome of the transaction (success/failure) and the return value(s) (if any) match the fault-free runs.

Regarding *integrity evaluation*, the key aspect involved is to verify the ledger integrity, which can be done by comparing the read/write sets of each transaction executed by a faulty contract with that of the reference contract. Integrity is maintained only if all key-value pairs in the read/write sets are exactly the same. In this work, the integrity verification is done using the transaction read/write set feature of Hyperledger Fabric [16], in a way that it neither interferes with workload execution nor requires additional instrumentation and potential change coverage checking (as getter functions would). Although such changes could be detected to some extent by getter functions on the contract, the read/write sets allow us to catch parasite side effects as well.

IV. EXPERIMENTAL EVALUATION

This section presents the implementation of the approach outlined previously. It explains how target smart contracts are selected and prepared for testing, how fault injection is implemented, how workload is generated and executed, how formal verification is executed, and how the test and formal verification results are analyzed.

A. SMART CONTRACT SELECTION AND PREPARATION

To the best of our knowledge, neither a definitive general taxonomy for smart contract functionality nor representative smart contract sets for security assessment, formal safety/liveness analysis, or dependability analysis exist yet. The existing business use cases and token taxonomies, while useful, do not provide meaningful source code template libraries. Thus, we started by selecting five commonly-used contracts (i.e., representative in this case), also with the intention of covering a large cross-section of smart contract use cases. For each of the five *base contracts*, we actually generate an unprotected version and a fully protected version of it, leading to a total of 15 smart contracts, in a process described in the next paragraphs. The five *base contracts* used as a basis for our experiments are the following:

- **State Machine:** The state machine contract [55] realizes a supply-chain use case, where a single product is tracked across multiple parties. A temperature and humidity sensor constantly reports readings about the conditions of the product. The contract tracks whether the transport conditions satisfy predefined compliance criteria. This contract is representative for the *permissioned state machine/business process* of the distributed ledger world. It is typically deployed on permissioned and private blockchains [37].

- **Wallet:** The wallet contract [56] realizes a simple Ether store, where users of the contract can deposit and withdraw Ether. It is representative of *bank-like "cryptocurrency wallet"* and refundable escrow functionalities from the cryptocurrency domain. It is typically deployed on unpermissioned and public blockchains. It is worth noting that its implementations include the reentrancy bug that is the cause of the DAO hack [11].
- **Token:** The token contract [56] implements the important aspects of an ERC20 standard-compliant fungible token [57], which has a limited supply and can be transferred between users, either individually or in batches. It is representative of a *very general pattern* found across smart contracts [58].
- **Refund Escrow:** The escrow contract [59] makes it possible for an escrower (the contract owner) to manage deposits incoming from multiple sources for designated payees. The contract also provides operational states like enabling/disabling refunding and withdrawal for a beneficiary user. It is *typically deployed on unpermissioned, public blockchains*.
- **Storage:** The storage contract [56] provides a *permissioned key-value storage* for users to store information encoded as integers, which is associated with their accounts. The contract has a designated administrator who has access to every storage slot regardless of ownership.

As mentioned, for each of the above 5 *base contracts*, we actually have the following different versions (described in further detail in the next paragraphs).

- **Base contract:** The original unmodified contract, just as described previously;
- **Stripped contract:** A modified version of the original contract, where protection mechanisms have been explicitly removed for evaluation purposes.
- **Protected contract:** A modified version of the original contract, where additional protection mechanisms have been added.

Each of the *base smart contracts* contains a "reasonable" level of defensive mechanisms (e.g., using `require` statements for error detection purposes), which stem from the business purpose of the contract (e.g., for input checking or access control), under the assumption that the implementation and their execution is fault-free. Solidity, as a language, provides assertion-style error detection mechanisms, and EVM error detection covers some basic situations as, e.g., array indexing out of bounds. For handling the detected errors, Solidity uses state-reverting exceptions, and EVM safely reverts all changes made to the state. In general, exceptions can emanate from the following situations: i) Invalid operations in-contract, such as misindexing arrays; ii) Errors with calling/creating other contracts and certain ledger operations; iii) Failing application-level checks using the `assert`, `require` and `revert` statements [14].

The *stripped smart contracts* were created to allow us to understand the impact of faults on contracts that lack

protection mechanisms. If an unprotected contract may be used to harm the overall system, it is essential to understand the potential impact of the presence of a specific software fault. The stripped smart contracts were created by simply stripping out the `require` statements from the base contracts.

Finally, we generated fully *protected smart contracts*, mostly as a way of understanding if such mechanisms could be sufficient to protect the contracts from certain types of faults. The protected smart contracts were created through the following two types of modifications. First, we include `assert` statements into each function as post-conditions (i.e., checked directly before the return of the "protected" functions). The assertion predicates were created manually, after carefully considering the valid and invalid function-local and contract states after execution. It is worth noting that none of the five selected contracts include `assert` statements. Second, where applicable, we extend each function to return part of its ledger side effect as a return value. Originally, most functions of the five selected contracts terminate without returning any values. The intention of these two modifications is to obtain a superior contract (in terms of error detection/mitigation) as, on the one hand, failed assertions suppress integrity-compromising state changes (at the expense of accessibility - failing calls do not modify the ledger); on the other hand, return values allow blockchain clients to perform error/failure detection themselves. In blockchain systems like Hyperledger Fabric, a client can choose to abandon a transaction *after* execution, but still *before* commitment. Note this latter case (i.e., error/failure detection by clients) may be only theoretically feasible, as, in practice, factors ranging from constrained resources to the need to use gateway-based blockchain access can prevent clients from cross-checking expected and actual ledger modifications.

Given the above descriptions, we call a base smart contract, its stripped version, and its protected version a **contract family**.

B. FAULT INJECTION IMPLEMENTATION

The faults defined in the fault model presented earlier in Section III-B are injected into all versions of the selected smart contracts resulting in multiple faulty versions of each original smart contract. A faulty contract will carry exactly one single artificially introduced fault. Smart contracts written in Solidity have to be compiled to bytecode using a compiler tool like Solc [60]. Solc generates an *abstract syntax tree* (AST) from the solidity source code, which can be easily manipulated to emulate faults. We have implemented fault injection as a series of transformations over the generated abstract syntax tree of the smart contracts, which is extracted from the Solc compiler [60].

Each fault is defined by a *condition* and an *action*. We read the AST, and whenever the specified condition holds for a given AST node, we transform it using the action. For example, to inject a Missing *if* construct around statement (MIA) fault, the condition checks whether the current node being

analyzed is an *if* statement. If the condition holds, the action replaces the node with its children (i.e., the body of the *if* statement).

Sometimes there are multiple candidates, within the same smart contract, that can be used for injecting a certain fault (i.e., multiple nodes matching the condition). Whenever this occurs, we generate one faulty contract for each of the matching locations, resulting in several mutated contracts (again, we emphasize that each contract will hold a single fault). After injecting the faults, we then serialize the ASTs back to Solidity source code so that it can be compiled to bytecode and deployed on the blockchain.

C. WORKLOAD GENERATION

During our experiments, we evaluated the feasibility of using automatically generated call sequences (i.e., a random workload). On the one hand, due to certain typical characteristics of Solidity smart contracts (e.g., only a few addresses actually having test-transferable tokens in deployed and test-initialized smart contracts), an automatic test case generator tends to create an unacceptably high ratio of calls that fail by specification. Thus, in this context, the high number of failing "garbage" transactions (due to their randomness) would shadow the interesting failures occurring in regular user-contract interactions. On the other hand, certain software faults can only be triggered by very specific input sequences. For these reasons, we manually defined a workload (i.e., a sequence of calls), exercising the target functions to observe the potential effects of the injected faults in every contract mutation. The construction of the workload was based on the combination of basic *black-box* and *white-box* testing principles.

Regarding the **black-box perspective**, we used the specification and the interface, to uncover the following aspects of each contract: i) the *actors* interacting with the contract; ii) the *classification of functions* based on whether they have side-effects on the ledger state or not, i.e., transaction and query functions, respectively; and iii) the *input parameter space* of individual functions. Based on these aspects of the contracts, and from a *black-box perspective*, the workload is generated as follows.

- Issue transaction function calls by mixing the actors and parameter value space (detailed in the following paragraphs).
- Issue the whole set of available query function calls. This mimics user-initiated self-check queries.
- For both transaction and query function calls, use multiple actor identities, starting with the actors who should not be allowed to perform the given action.
- When a function has input parameters, use *equivalence partitioning* and *interval testing* principles to exercise the function with meaningful inputs. Equivalence partitioning typically splits the domain of an input parameter into valid and invalid partitions (derived from the business logic), allowing testers to pick a single value from each partition as a representative input.

Interval testing picks additional values from the partitions/ranges, usually from around their boundaries (if defined) to catch off-by-one errors. The combination of the two approaches reduces the number of required function calls for building a representative test sequence.

We also considered a **white-box perspective**, whenever the contract exhibited a state-machine-like (or workflow-like) behavior. We first construct the state transition graph of the contract by inspecting its implementation. The transitions in the graph drive the test sequence construction to achieve the highest possible state and transition coverage. Note that the complete coverage by a single test sequence is impossible if the state transition graph of the contract has multiple terminal strongly connected components (e.g., multiple sink states).

It is worth mentioning that, for the sake of readability, we do not differentiate between transaction and query function calls in the remainder of the paper. Note, however, that query function calls do not produce a read-write set and only the endorsement phase of the Fabric consensus applies to them.

D. TEST HARNESS AND EXECUTION

Our experimental environment is a highly instrumented Hyperledger Fabric lab deployment, equipped with the Ethereum Virtual Machine (EVM) implementation of Hyperledger Burrow. The environment uses Hyperledger Caliper for most experiment automation tasks. We use the environment to perform *test runs*: *executing the contract family workload (test sequence) of a specific mutation*. In our environment, the execution of a test run follows the order below.

1) The system is (re)initialized, meaning that the Hyperledger nodes (endorsers and orderers) are set (or reset) in a clean initial state, with an empty blockchain and the Burrow EVM support installed. Then, the contract under assessment is deployed onto the endorsing peers and the appropriate accounts are filled with local Ether to provide transferable cryptocurrency for certain contracts.

2) For each transaction in a test sequence of a contract family:
    a) The (Caliper) client sends the transaction request to each endorsing peer.
    b) The endorsing peers forward the call to the EVM implementation, running fenced-off in a Docker container.
    c) If the EVM execution terminates without error, the read-write sets are sent back to the client. Otherwise (upon EVM error, or if the Fabric peer terminates processing after the specified timeout) the client receives an error signal.
    d) If all endorsements permit (i.e., n-out-of-n consensus), the client sends them to the ordering service, which immediately creates a one-transaction block and distributes that to the peers for *commitment* (inclusion in the current ledger state). It is worth noting that, while strictly sequential

TABLE 3. Contract statistics by families and variants.

Contract Family	Test Length	Variant	Asserts	Requires	Formal Specifications	LOC
State Machine	252	Base	-	10	35	181
		Stripped	-	-	35	124
		Protected	7	10	35	198
Wallet	19	Base	-	2	6	28
		Stripped	-	-	6	27
		Protected	3	2	7	39
Escrow	183	Base	-	9	31	243
		Stripped	-	-	31	219
		Protected	8	9	33	265
Storage	72	Base	-	5	13	67
		Stripped	-	-	13	62
		Protected	8	5	13	83
Token	151	Base	-	6	10	72
		Stripped	-	-	10	43
		Protected	6	6	10	92

transaction execution with one-transaction blocks and n-out-of-n consensus are not production-typical for Hyperledger Fabric, for our purposes these are admissible and useful simplifications.

e) The client proceeds with the next transaction only if the pending has been committed or can be deemed failed.

During the test runs, detailed *transaction-level* data is collected from multiple sources. Caliper/client-side observations provide the majority of transaction details, such as timing of life-cycle phases, side effects returned by the platform and general status information, for example, errors in case of unsuccessful endorsement. Low-level EVM runtime data is also collected from contract-side logs.

The collected data set is post-processed to match and compare the transactions of faulty contracts to their corresponding transaction pair in the reference contracts. The key comparison points are detailed in Section IV-G.

E. FORMAL SPECIFICATION AND VERIFICATION

For performing the formal verification, we execute a formal verification tool, namely solc-verify [36], over the smart contracts. Solc-verify is an automated modular verifier for Solidity smart contracts. It takes contract source code annotated with *specifications* (e.g., contract invariants, pre- and post-conditions) and discharges verification conditions using modular program verification and SMT solvers. Solc-verify targets the *functional correctness* of contracts and checks for possible assertion violations (Solidity `assert` statements) and overflows as implicit specification. Moreover, it has a feature for additional specifications such as:

- *Contract invariants* that must hold after the constructor, and before and after every transaction;
- *Function pre- and post-conditions* that are assumed before the function and checked after the function;
- *Loop invariants* to specify loops with expressions that must hold before, after and in every iteration;
- *Modification specifiers* for fine grained specification of the contract state that a function can modify.

In this work, for each contract family, a single set of specifications is created and used to annotate the base contracts, the stripped, and the protected contracts. The specifications are created manually for each base contract, using the facilities (invariants, preconditions, etc.) described in Section III-E and [36]. Table 3 presents key statistics about the number of assert and require/revert statements, and the formal specification constructs for each family.

Solc-verify translates each smart contract into an intermediate verification language [53] and performs modular verification [52], i.e., for each function it is checked whether its specification (e.g., invariants, post-conditions, assertions) hold, given the pre-conditions. The main feature that makes modular verification efficient is that when a function calls another one, the callee is replaced with its specification. However, this might introduce false alarms if the specification is not detailed enough. Note that, as solc-verify targets functional correctness (i.e., safety property and not liveness [36]), a contract that always reverts is considered correct (or safe) by the verifier.

F. VERIFICATION AND PROTECTION LEVELS IN SMART CONTRACTS

Our experiments resemble, from a verification perspective, the life-cycle of a smart contract. Thus, some of the injected faults might be captured at different levels. In our experimental setup that is using Hyperledger Fabric running Solidity smart contracts using the Hyperledger Burrow EVM, this translates to the following 4 protection levels (i.e., including formal verification). Note that names in bold denote the terminology used in the figures presented in next section.

- **Formal verification**: applied before deployment (e.g., static analysis), it may detect a certain portion of existing faults *before*.
- **Contract self-check**: specification-derived defensive programming constructs in the source/byte code (e.g., `require` statements in Solidity) and additional protective measures present in the source code (e.g., error

detection by invariant checking directly before function return using `assert` statements).

- **Runtime platform check**: error detection mechanisms in the EVM runtime implementation (e.g., detection of array out of bounds indexing, insufficient Ether balance for transfer requests, invalid address format, or runaway execution depleting gas) and error detection mechanisms in Hyperledger Fabric – execution attempt timeout detection and result matching over the endorsing peer set. Both contract self-checks and runtime platform checks provide *error detection and processing*; and in our case, limited fault tolerance through what is effectively a *rollback* of the affected smart contract call.

- **Output invariant check**: a blockchain client may be also able to detect an execution error based on the return values it receives (assuming the availability of an appropriate test oracle). In general, it has, at least, failure detection capabilities (after a transaction has been committed to the ledger), given the necessary output-based observability and test oracle. Additionally, in a blockchain system where the client can also be involved in transaction consensus – as is the case with Hyperledger Fabric – the client can also choose to *abort* the transaction attempt.

G. RESULT ANALYSIS

As described above, each faulty contract is executed against its family-specific workload. Contract execution was performed exactly once for each test case; repeating the experiments showed a very high level of determinism (as expected, slight differences arise only in timing and "runaway" execution terminations).

The guiding principle of our assessment of the test case executions is that we aim at comparing behavior during executions on a faulty contract versus its corresponding fault-free reference. For instance, the execution behavior of a faulty protected contract should be compared to the execution behavior of the fault-free protected contract, and for the same workload. Analysing runtime behavior of the faulty contracts without comparing it to the corresponding reference run would not be meaningful as some test sequence steps are intended to fail also in baseline and protected cases (e.g., authorization checking).

In each execution, *a transaction is deemed successful* if and only if i) all of its endorsements are successful and matching and ii) it is successfully ordered and reported as committed by all endorsing peers. Additionally, formal verification was performed for each faulty contract, taking into account the supplied, contract family-specific verification properties as well as the `require` and `assert` statements present. The resulting observations that we subject to data analysis include the following key characteristics and error/failure model for each faulty contract version

- **Observation identifier**: which includes *contract family, version (i.e., base, stripped, protected), fault ID, mutation instance ID*.

- **Source code formal verification result**: which indicates whether the fault-free and faulty contracts were successfully distinguished by the formal verification tool. The result is reported as *true positive*, *true negative*, *false positive*, or *false negative*. Note that the *true positive verification results* indicate the caught faulty mutations. Contract execution results are important for the remaining cases (the false and true negatives).

- **Abort failure**: the result is a Boolean (true or false) that indicates whether there was at least one transaction in the faulty contract that was aborted, while the reference case did not fail. *Abort failure* shows that contract or platform checks detected at least one transaction invocation failure correctly. Specific error codes [61], which are not documented here, allow us to distinguish between the contract-level and platform-level checks' success or failure. Note that failing "normal" checks (e.g., `require` statements) can also be distinguished from the source code fault indicating error detecting ones. This failure is detectable by the client.

- **Gas depletion or Fabric timeout**: it indicates that a transaction call in the faulty contract failed through a *rollback* of the affected smart contract call due to gas depletion or Fabric timeout, while it did not happen in the reference run. Gas depletion and Fabric timeout respectively occur by EVM runtime implementation and error detection mechanisms in Hyperledger Fabric protection mechanisms.

- **Reliability failure**: the result is a Boolean and indicates whether there was at least one transaction in the faulty contract with a different result or return value than the reference contract. *Reliability Failure* indicates (theoretically) client observable failures during output invariant checks.

- **Integrity failure**: the result is a Boolean that indicates whether there was at least one successful transaction in the faulty contract with a different result or return value and a different read-write set (i.e., the faulty contract modified the state of the blockchain differently from the reference contract) than the reference contract. In this situation, the ledger integrity failure is also detectable by the client.

- **Latent integrity error**: the result is a Boolean that indicates whether there was at least one successful transaction in the faulty contract with the same result or return value but a different read-write set than the reference contract. In this situation, the ledger integrity error stays hidden and cannot be directly detected by the client as the client receives the expected result or return value.

- **Ineffective**: the result is a Boolean that indicates whether the result, return value, and read-write set of all transaction calls of the faulty contract are same as of the reference contract.

In fact, faulty contracts passing even output invariant checks constitute cases where the injected fault remains **undetected** – i.e., no fault-indicative error state is detected.

Fault effect	Transaction not concluded	Incorrect return value or transaction result	Incorrect ledger state
Abort failures	✓		
Gas depletion/Fabric timeout	✓		
Reliability failures		✓	
Integrity failures		✓	✓
Latent integrity error			✓
Ineffective			

These undetected faults are either *ineffective*, or, on the contrary, constitute *latent integrity errors*.

The error/failure model, including the effects of smart contracts faults, and its characteristics are summarized in Table 4. In this table, to simplify, the effects of fault are categorized into three groups: i) Transaction not concluded; ii) Incorrect return value or transaction result; iii) Incorrect ledger state. When an *Abort failure* occurs, for any reason [61], neither a value or transaction result is returned to the client, nor any changes are made in the state of ledger. The transaction is simply not concluded and some sort of exception or error message is given to the client allowing to detect the failure. When a *Gas depletion or Fabric timeout* occurs, same as *Abort failure*, no value is returned and no changes to the ledger state are made. In this case, the fault causes spending more resources in terms of time and gas. Thus, the transaction is not concluded due to the lack of gas or system timeout. With *Reliability failure*, the transaction is concluded but the return value or transaction result is incorrect, although the state of the ledger is still correct. In contrast, when an *Integrity failure* occurs, in addition to incorrect return value or transaction result, the integrity of the ledger state is violated as well. Either the lack of a valid result (in the case of *Abort failure* and *Gas depletion or Fabric timeout*) or an incorrect result (in the case of *Reliability failure* and *Integrity failure*) allows the client to detect the above issues. While in the case of *Latent integrity error*, the integrity of the ledger state is violated but, since the return value to the client is correct, it remains undetected by the client. A fault is *ineffective* when the faulty contract behaves exactly same as the reference contract.

V. RESULTS

This section presents the results obtained during our experimental evaluation. It first overviews the effects caused by the different injected faults. Then, we evaluate the effectiveness of formal verification and other protection mechanisms. Finally, we describe the preliminary performance evaluation results.

A. FAULT EFFECTS OVERVIEW

We begin by presenting the distribution of injected faults and their impact in terms of the error/failure model, previously defined in Section IV-G, and with respect to the base contracts only, as they are the most realistic versions. Fig. 3 to Fig. 6 show the distribution/impact of the faults, listed from the most frequent to the least frequent ones in the x-axis. For instance,

Wrong Algorithm (WALR), which is the most frequent fault, is present in many of the faulty contracts and *Wrong Arithmetic Expression Used in Assignment (WVAE)* is present in just a few faulty contracts. The frequency of the faults presented in Fig. 3 to Fig. 6 does not directly imply the frequency of the mistakes that developers usually make but relates to the number of possible places in the typical contracts' code, in which each type of fault may appear. The faults that are not present in the figures did not appear in any base contract. The very first observation is that, in general, the generic faults are more frequent than smart-contract specific faults.

Each of the four plots shows the number of times (i.e., the number of faulty contracts with a specific fault) each individual fault caused a specific problem regarding the error/failure model. For instance, Fig. 3 shows that from the 53 base faulty contracts with the *Wrong Algorithm (WALR)* fault, *abort failure* occurred in at least one of the transactions executed against 14 faulty contracts (red part in the figures), and as shown in Fig. 4, *reliability failures* occurred in at least one of the transactions of 10 faulty contracts, and so on. Note that each fault may cause more than one failure and error in the same contract. In total, 71 injected faults caused *abort failures*; 145 injected faults caused *reliability failures*, 122 injected faults caused *integrity failures*; and 90 injected faults caused *latent integrity errors* in at least one of the transactions executed against the faulty contracts. Thus, reliability failure is the most frequent failure, and abort failure is the least frequent one.

As we can see in Fig. 3 to Fig. 6 *Missing function call (MFC)* is highly frequent and most errors and failures, specially *reliability failures* and *integrity failures*, are caused due to this fault (a total of 82 failures/errors). *Latent integrity error* that is the most serious issue, is mostly caused by *Missing variable assignment using an expression (MVAE)* and *Missing function call (MFC)*. On the other hand, *Missing require OR subexpression on input variables (MROIV)* is the only one that did not cause any failure or error; it was not a frequent fault in base contracts and when happened, did not cause any failure or error. *Wrong arithmetic expression used in assignment (WVAE)* is not frequent either but, when occurred, caused all kinds of failures and errors.

Wrong logical expression in parameters of function call (WLEP) is present in 16 faulty contracts and in all cases triggered a reliability failure. *Missing return statement (MRS)*, *Missing if construct around statement (MIA)*, and *Wrong*
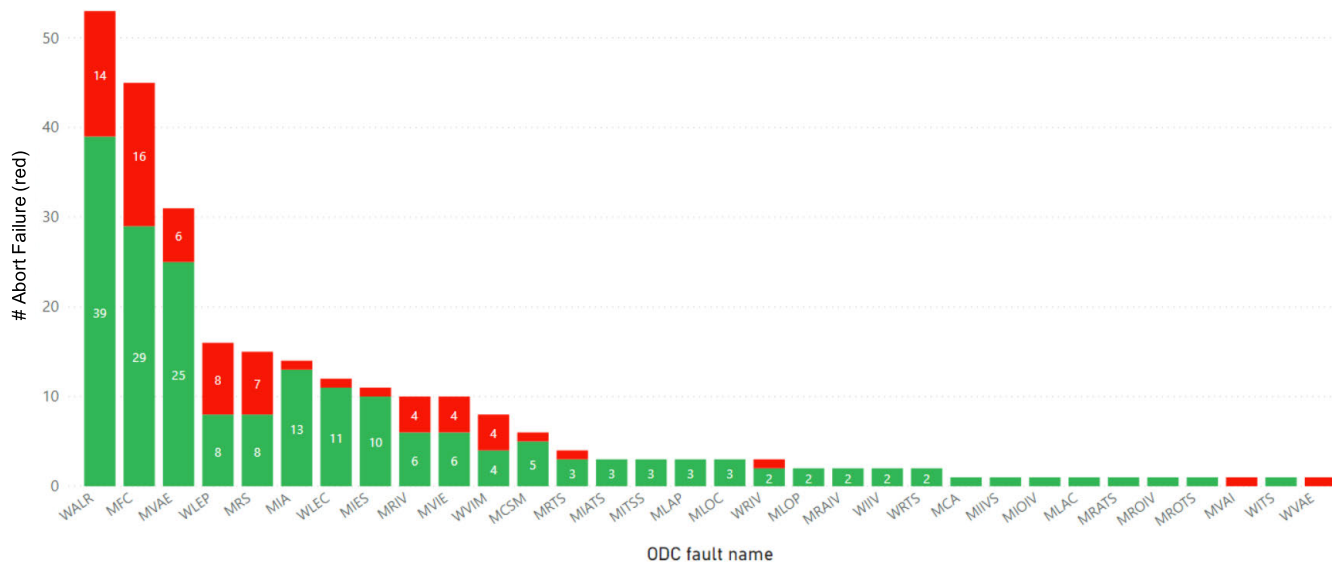
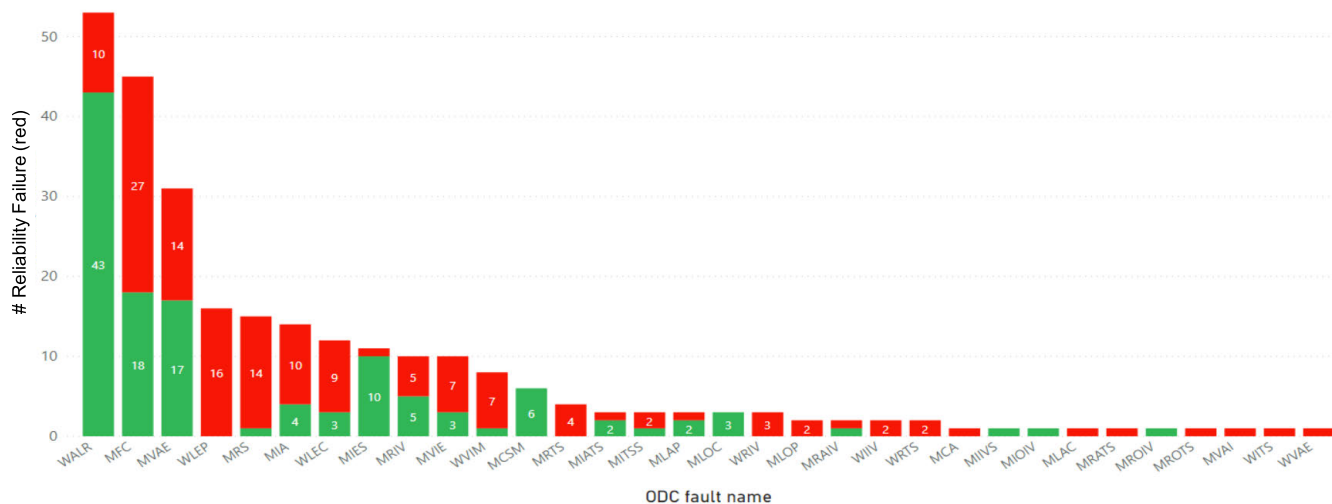**FIGURE 3.** Faults' impact in terms of abort failure.

FIGURE 4. Faults' impact on reliability.

logical expression used as branch condition (WLEC) also mostly caused reliability failures.

Among smart contract-specific faults, *Missing require on input variables (MRIV)* and *Missing call to SafeMath (MCSM)* are more frequent faults. *Missing require on transaction sender (MRTS)*, *Wrong logical expression in require over input variables (WRIV)*, *Wrong logical expression in if construct over input variable (WIIV)*, and *Wrong logical expression in require over transaction sender (WRTS)*, although not being very frequent, caused reliability and integrity failures in all faulty contracts in which they were injected.

B. BASE CONTRACT RESULTS

We visualize the sequence of successively escaping faulty contracts using Sankey-like diagrams. In these diagrams, blue rectangles represent protection mechanisms, and ellipses are classification of system output, referring to cases where a given contract was indeed labeled as faulty at a given phase. The order of steps is quite typical, representing a suggested development workflow and the protection logic of EVM, while the thickness of the lines shows the ratio of contracts passing a check. First, we analyze the detection results for all faulty instances of the base contracts, measured as described earlier (refer to Section IV-F and Section IV-G).
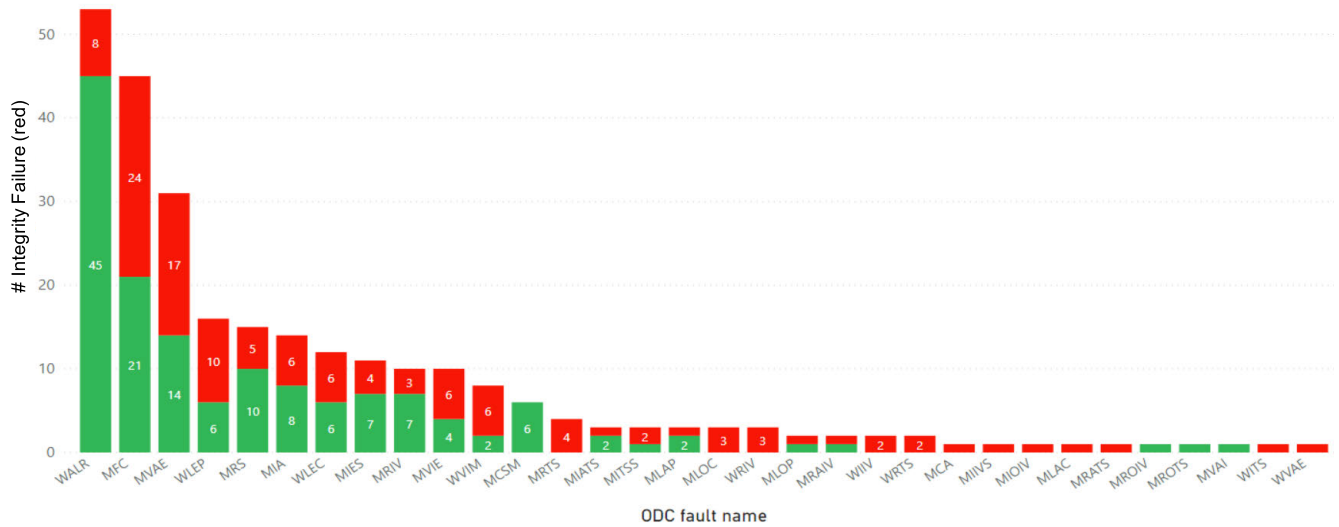
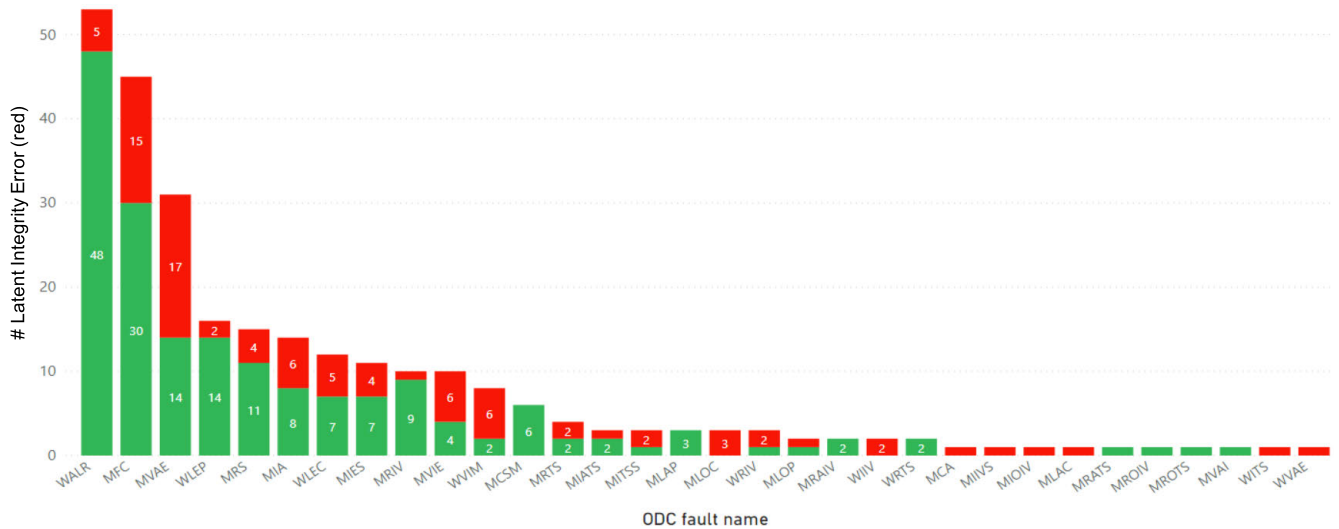**FIGURE 5.** Faults' impact on ledger integrity.

FIGURE 6. Faults' impact in terms of latent integrity error.

Fig. 7 shows the number of faulty base contracts eluding the different detection phases, but assuming that no formal verification is performed before deployment, while Fig. 8 shows the number of faulty contracts escaping detection at the different phases, which include verification as part of the workflow. The number of generated faulty base contracts is 268, which is visible at the left side of Fig. 7 and Fig. 8.

When verification is part of the workflow (see Fig. 8), all faulty contracts are first subject to formal verification, which detects 159 faults out of 268 (59% of all faulty contracts). The next step, contract-level self-check, expressed in the form of require and assert constructs, catches faulty contracts where a transaction terminated with an endorsement result different from the reference run. Contract-level self-check detects 37 out of 110 faults (which is 14% of all 268 faulty contracts) that remained undetected by formal verification.

The next step involves platform-level runtime checks. These checks refer to built-in EVM and Hyperledger error detection mechanisms (e.g., detection of a timeout, out-of-bound array addressing, etc.). Violating such checks corresponds to a behavior, in which a non-FT environment could result in severe failures, including system crashes. In the current experiments, no faulty contracts were detected by these checks.

The last level of protection mechanisms is the client-side validation of results (i.e., output invariant check in Fig. 7 and Fig. 8). In total, 13 out of 73 faults (5% of all faulty contracts) that remained undetected by previous verification and checks were detected in this final step. In general, the remaining faults which are undetectable by this step either cause **latent integrity errors** or are **ineffective**. The first case represents the most critical class of failures referring to a contract which
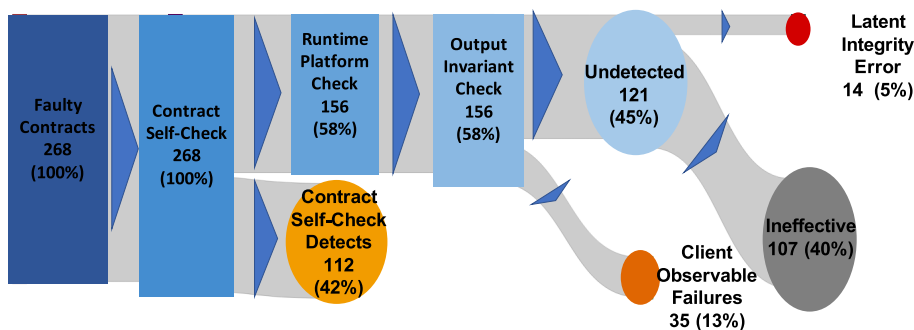
FIGURE 7. Fault detection in *base contracts* without formal verification.
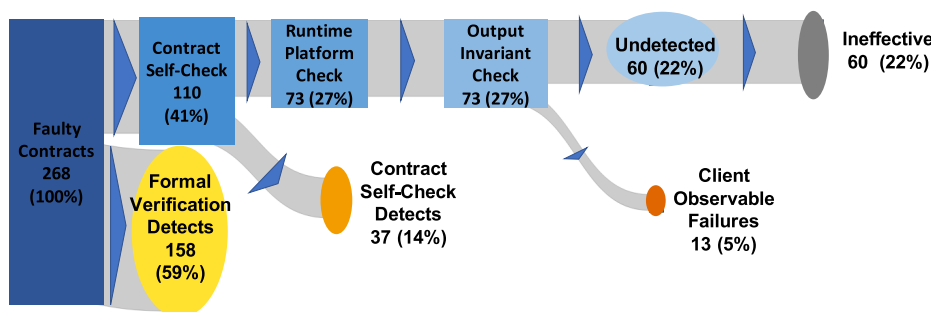

FIGURE 8. Fault detection in *base contracts*, with formal verification.

may produce corrupted, uncovered ledger content, while mutants having only ineffective faults cannot be distinguished from the reference (i.e., their injection, in combination with the workload, did not produce any change in the contract behavior). In the case of baseline mutants with formal verification (refer to Fig. 8), all 60 remaining faults (22% of all mutants) are ineffective.

The results, presented in Fig. 7 and Fig. 8, suggest that for contracts with ''reasonable'' protections (i.e., the *base contracts*, which apply simple defensive programming techniques), verification and runtime checks are complementary techniques for detecting faulty contracts with potentially effective faults. On the one hand, formal verification applied to the faulty contracts caught only 59% of the faults. On the other hand, verification does filter out those 14 faulty contracts (5% of all mutants), which led to latent integrity errors without verification (see Fig. 7).

Faults that are not ineffective, but pass verification and self-checks are typically related to liveness or insufficient specification. In the former case (liveness), faults cause to revert transactions that should not revert. Formal verification currently targets safety property (and not liveness) and does not detect such issues. Furthermore, self-checks have no chance to catch these issues as the transactions are reverted before the checks. In the latter case, the specification was not sufficient enough, e.g., for the state machine contract, we did not specify the expected behavior for all the (roughly 20) state variables.

One particularly interesting case escaping both verification and self-checks was the missing auto increment for loop variables. This caused an infinite loop that was forcefully terminated by the ledger. Catching this error in Solc-verify requires reasoning about termination (which is complicated and not supported, at the time of writing), and catching them by self-checks would require complex assertions on the progress of loops.

Note that latent integrity errors are arguably the worst failure mode for distributed ledgers. As ledgers are essentially built over distributed databases, their generally expected operational model explicitly rules out transaction rollback. As a consequence, to repair data corruption, the only viable options are governance processes (by blockchain peer consortium) and embedding compensatory mechanisms into the on-chain workflows.

We deem the significant ratio of ineffective faults normal. Solidity smart contract testing has to deal with the vast Ethereum address space; until such time that symbolic automated test pattern generation becomes available for Solidity and/or the EVM, the only viable option is to, as we did, manually define the workload to be used. Evaluating test coverage is part of further research.

Fig. 9 presents the results of the effectiveness of formal verification and fault detection mechanisms in the faulty base contracts, by showing in which detection phase the injected faults are detected. Results show that *Missing function call (MFC)* and *Missing variable assignment using an expression (MVAE)* that causes more severe failures, and errors
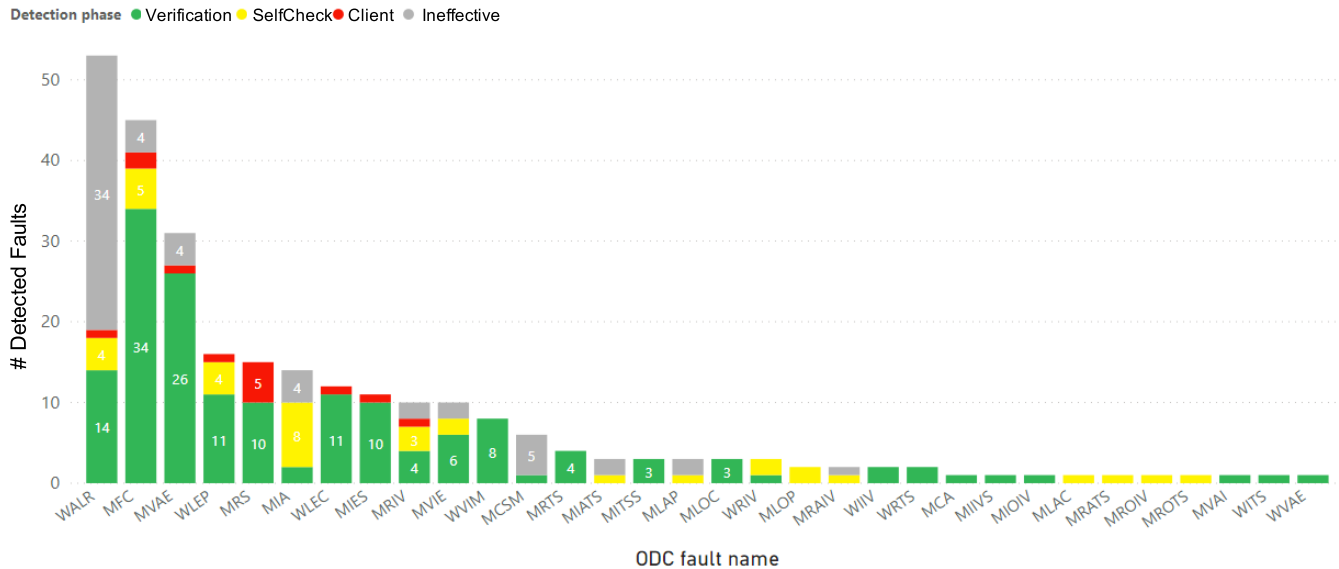
FIGURE 9. Effectiveness of formal verification and protection mechanisms on each fault in the *base contracts*.

are mostly detected by formal verification. Similarly, *Wrong logical expression in parameters of function call (WLEP)*, *Missing return statement (MRS)*, and *Wrong logical expression used as branch condition (WLEC)* (that cause reliability failures in most cases) are also mostly detected by formal verification. However, formal verification does not seem to be very effective in cases like *Missing if construct around statement (MIA)*. On the other side, contract self-checks appear to be effective in detecting this fault.

Among all faults, *Missing return statement (MRS)* is the one that, after applying formal verification and protection mechanisms, causes more reliability/integrity failures, but is detectable by the client (red bar in the figure). Among other problematic smart contract-specific faults, *Missing require on transaction sender (MRTS)*, *Wrong logical expression in if construct over input variable (WIIV)*, and *Wrong logical expression in require over transaction sender (WRTS)* are fully detected by formal verification. However, the same does not happen with *Missing call to SafeMath (MCSM)* and *Wrong logical expression in require over input variables (WRIV)*.

The final general observation is that, after applying verification and protection over the base contracts, only a few faulty contracts (13 out of 268, shown in red) cause reliability or integrity failures and all these cases are detectable by the client. Obviously, the difficulty lies in applying the correct mechanisms at the correct time, and fault injection may help in this regard by identifying cases for which developers must pay special attention.

C. PROTECTED CONTRACT RESULTS
Fig. 10 and Fig. 11 show the number of faulty *protected contracts* escaping detection at the different phases (with a

total of 279 protected faulty contracts). Fig. 10 refers to the case where formal verification is used, whereas in Fig. 11 verification was skipped. The different number of faulty contracts is due to slight differences in source code. While `assert` statements are not subjected to faults, their supporting modifications are.

In comparison to the *base contracts'* case presented in the previous section, the following three main findings can be formulated.

- Assuming no verification, *even "cheap" and simple post-condition checking (on-chain and client) significantly reduce the ratio of latent integrity errors*, and the significance of client-side checks is slightly reduced.
- Verification becomes significantly more effective; it can use the protective statements as additional implicit specification. However, we also experienced false positives for the Token contract, due to the verifier not being able to prove assertions in a loop.
- Combined verification and runtime checks eliminate latent integrity errors in both cases, and could also reduce the number of client observable faults.

Fig. 12 illustrates the results of the effectiveness of formal verification and fault detection mechanisms for the protected contracts. The types of faults that could be injected into the protected contracts are exactly the same as the faults that are injected into the base contracts, although their order in terms of frequency is slightly different due to the specificities of the additional protection mechanisms present in the code.

After analysing Fig. 12 and Fig. 9, we observed a very similar pattern regarding the effectiveness of the formal verification and protection mechanisms across these two groups of contracts. However, after applying verification and protection to the protected contracts, a lower number of faulty contracts
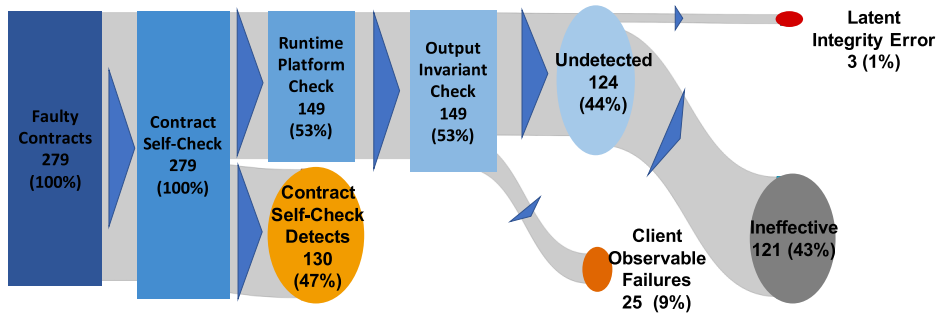
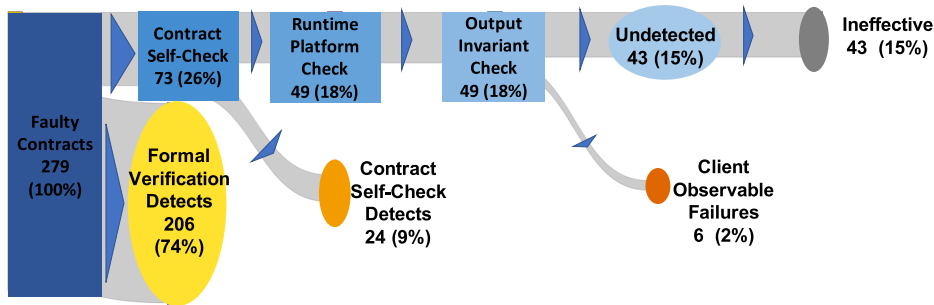**FIGURE 10.** Fault detection in *protected contracts* without formal verification.

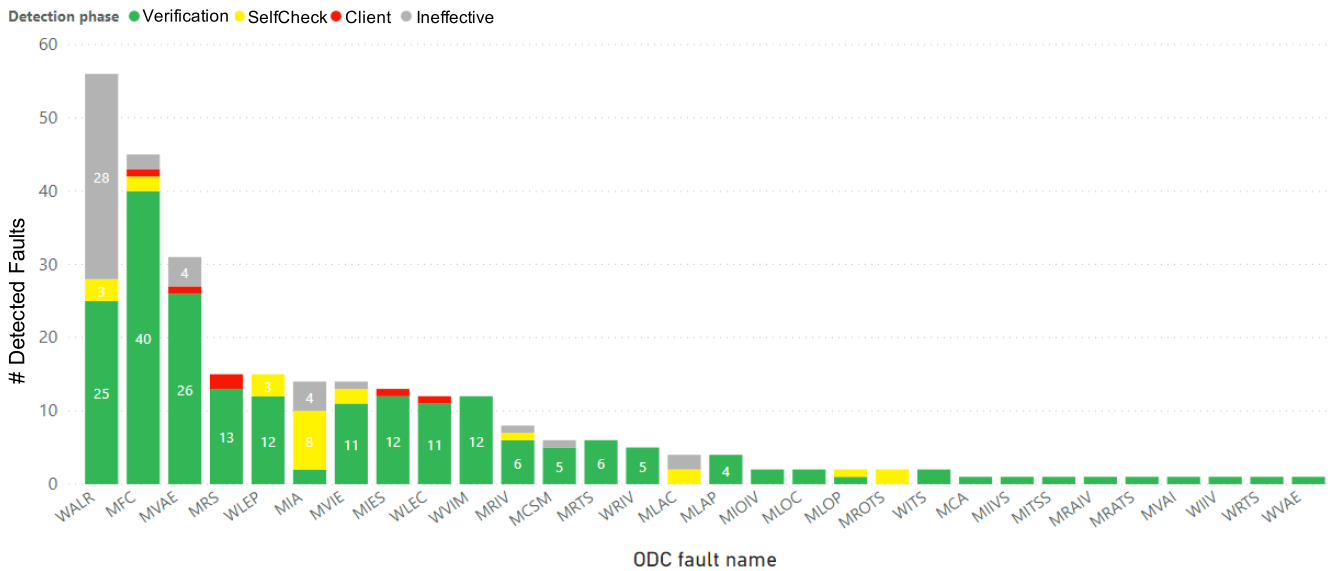**FIGURE 11.** Fault detection in *protected contracts* with formal verification.

FIGURE 12. Effectiveness of formal verification and protection mechanisms on each fault in the protected contracts.

(6 out of 279, shown in red) caused reliability or integrity failures (detectable by the client), when compared to the base contracts.

D. STRIPPED CONTRACTS RESULTS

Fig. 13 and Fig. 14 show the number of faulty *stripped contracts* escaping detection at the different phases (out of 104 stripped faulty contracts), without and with

formal verification, respectively. As shown, for stripped contracts without verification, more than half of the mutants results in client observable failures, as contract self-check and some built-in EVM mechanisms (like balance check) can catch only a few of the injected faults (10% in total). From the remaining faulty contracts, 13% of them result in latent integrity error, which is quite high. Formal verification, on the other hand, catches 97% of the faulty
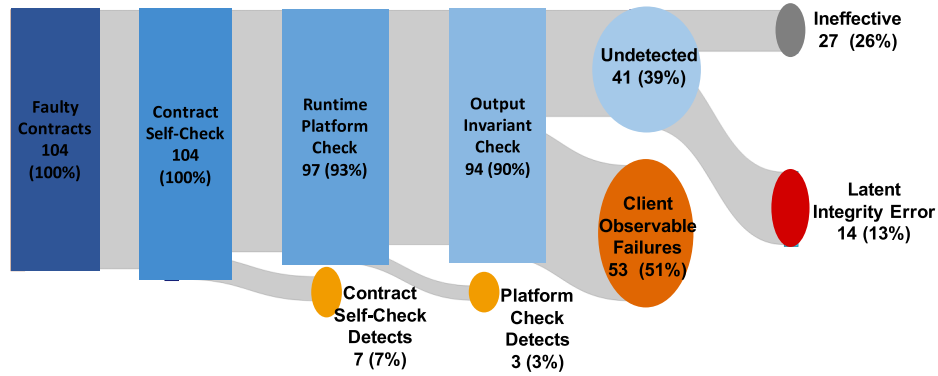
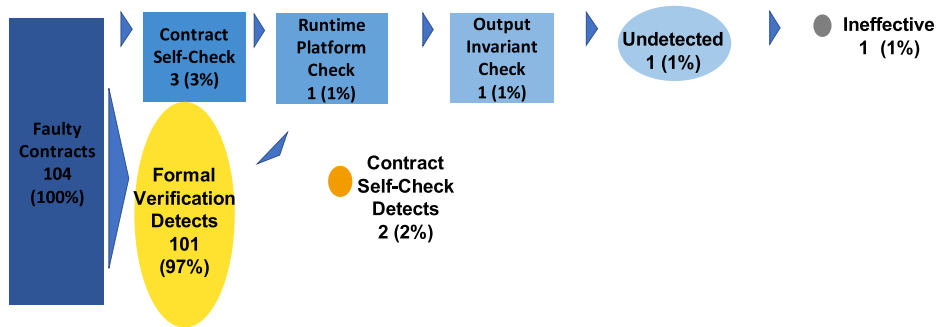**FIGURE 13.** Fault detection in *stripped contracts* without formal verification.

FIGURE 14. Fault detection in *stripped contracts* with formal verification.
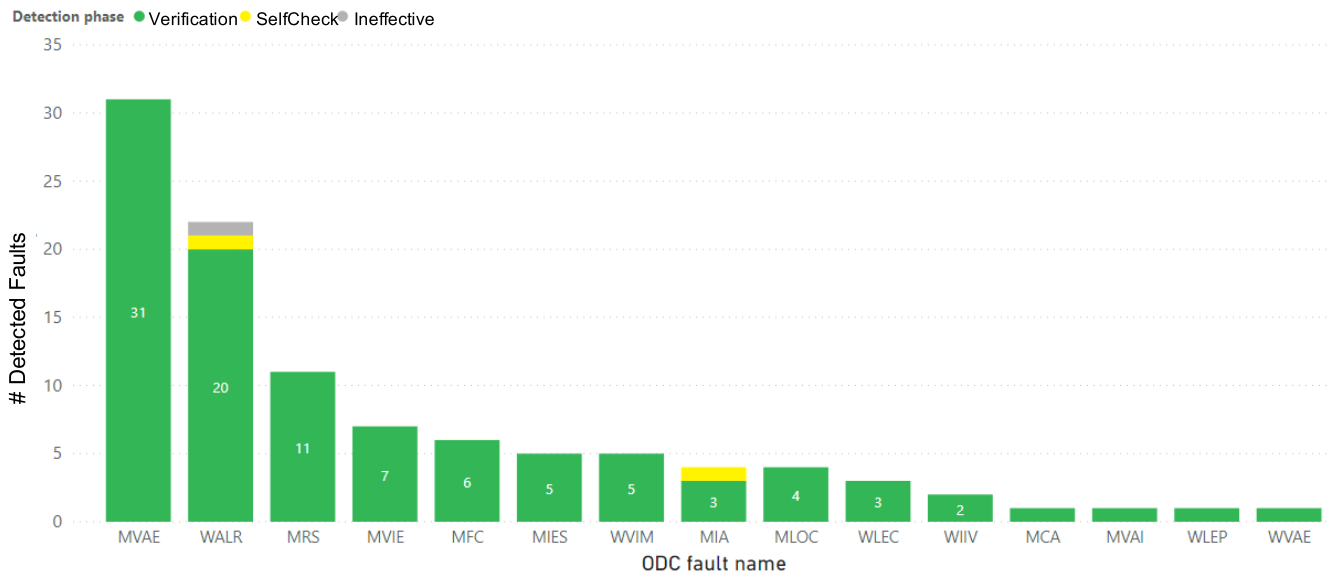

FIGURE 15. Effectiveness of formal verification and protection mechanisms on each fault in the stripped contracts.

contracts, and only one (ineffective) fault passes all protection mechanisms.

The results, on the one hand, confirmed our presupposition that eliminating even the basic runtime precondition checks leads to unacceptable levels of latent integrity errors. On the other hand, as we used the same precondition specifications

for all contracts in a family, verification could trivially find counterexamples in all cases due to the missing `require` statements.

The above observation is confirmed in Fig. 15, which presents the results of the effectiveness of formal verification and fault detection mechanisms for the stripped contracts.

As we can see, a smaller number (i.e., when compared to the base or protected contracts) of types of faults could be injected into stripped contracts. The main aspect visible is that almost all cases are detected by formal verification.

E. VERIFICATION EFFECTIVENESS

During our experiments, we observed latent integrity errors, but only when formal verification was not applied (i.e., 3 latent integrity errors in the protected contracts and 14 in the base contracts). We investigated all cases and came to the conclusion that all latent integrity errors were caused by missing constructs, for example missing variable initializations or missing subexpressions in the conditions of `if` statements. While the errors could have also been avoided with more protection, we observed that writing formal specification was, in some cases, more efficient than writing extra protection. For instance, instead of defining invariants individually at the function level, contract-level constructs can also be applied [36].

To evaluate the effectiveness of verification and the impact of additional protections, we summarized the results by comparing the *recall* values of base and protected variants. Recall is calculated by dividing the number of detected faulty contracts (true positives) over the total number of faulty contracts (the sum of true positives and false negatives). Table 5 presents the "verification recall", evaluated for different groups of faulty contracts with common error/failure effects (e.g., faulty contracts causing ledger integrity violation, faulty contracts causing reliability failures).

TABLE 5. Verification recall for faulty contracts with different error/failure effects.

Faulty contract set by error/failure effect	Verification recalls for base variants	Verification recalls for protected variants
Abort failures	69%	79%
Gas depletion/Fabric timeout	100%	100%
Reliability failures	67%	81%
Integrity failures	78%	84%
Latent integrity error	83%	76%
Ineffective	69%	81%
All effective faults	69%	81%

Results show that runtime protections added to the protected contracts, acting as additional specification, improve the effectiveness of formal verification (i.e., recall increases) in almost all cases. Hidden side effects are the exception. The reason for the lower recall, in this case, is that the "return value introduction" protective measure changes the "hidden side effects" baseline failure category of a number of mutants through increased observability (as also expressed in Fig. 8). Ratio-wise, the remaining cases of the protected mutants (no new ones introduced) become harder to catch with verification.

Due to restrictive default Burrow settings in the measurement setup, Fabric timeouts (1 second) were never engaged; thus, the "EVM gas depletion or Fabric timeout" cases actually only ever exercised the EVM mechanism that limits the number of computational steps. While verification caught all

such cases (e.g., infinite loops), we performed a dedicated experiment to also measure active Fabric-level protections, which is presented in the following section (Section V-F).

F. PRELIMINARY ASSESSMENT OF BLOCKCHAIN PERFORMANCE

As a complement to the experimental evaluation, we carried out a set of additional experiments of anecdotal nature, aiming at understanding performance issues related to the execution of faulty contracts. *Performance evaluation* allows us to understand if there are significant changes in the execution time of faulty smart contracts when compared to their reference counterparts. Protection mechanisms defined in the blockchain system (e.g., the definition of gas limit) do not allow smart contracts to execute indefinitely. Hence, performance problems can eventually be transformed into correctness problems (i.e., failed transactions). Additionally, and especially in permissioned systems, the different resource usage profiles of faulty contracts may have an impact on overall blockchain system performance and availability through resource contention. In this work, performance evaluation is mainly concerned about the execution time of transactions.

We performed our experiments using the Token *base contract*, with an infinite loop fault in its `batchTransfer` function and no faults in its `transfer` function (standard token management functions). The test sequence calls i) the fault-free function 400 times (warm-up fault-free period) to be used as reference and baseline behavior of the system for the comparison; ii) the faulty function 200 times; and iii) the fault-free function 400 times again to be compared with reference runs in order to understand how the faulty runs impacted on the execution time of the fault-free function. The calls were all data-independent. Burrow was reconfigured to enable execution to hit (and possibly far exceed) the Fabric timeout, but the measurement environment otherwise remained unchanged. The numbers 400 and 200 are arbitrarily chosen but in a way that lets us observe the behavior of the system long enough.

Fig. 16 shows the client-perceived transaction times as well as the call execution time measured *inside the EVM* through additional instrumentation. The colors distinguish the two functions: red for the faulty function and blue for the fault-free function. As illustrated by the figure, upon switching to the faulty function, end to end transaction time increases to around 1 s (the Fabric timeout), and transactions fail; however, when the workload switches back to the fault-free function, for approximately 14 seconds, it remains at 1 second before regaining the nominal value (and transactions begin to succeed). At the same time, EVM executions are not *actually* terminated in the containers, leading to increased resource contention.

Evidently, this is a resource management bug in the blockchain environment (CPU-overloading runoff executions are not terminated and garbage-collected in a timely manner). The significance of this phenomenon in our context is that it demonstrates that, in general, we *cannot* blindly rule out the
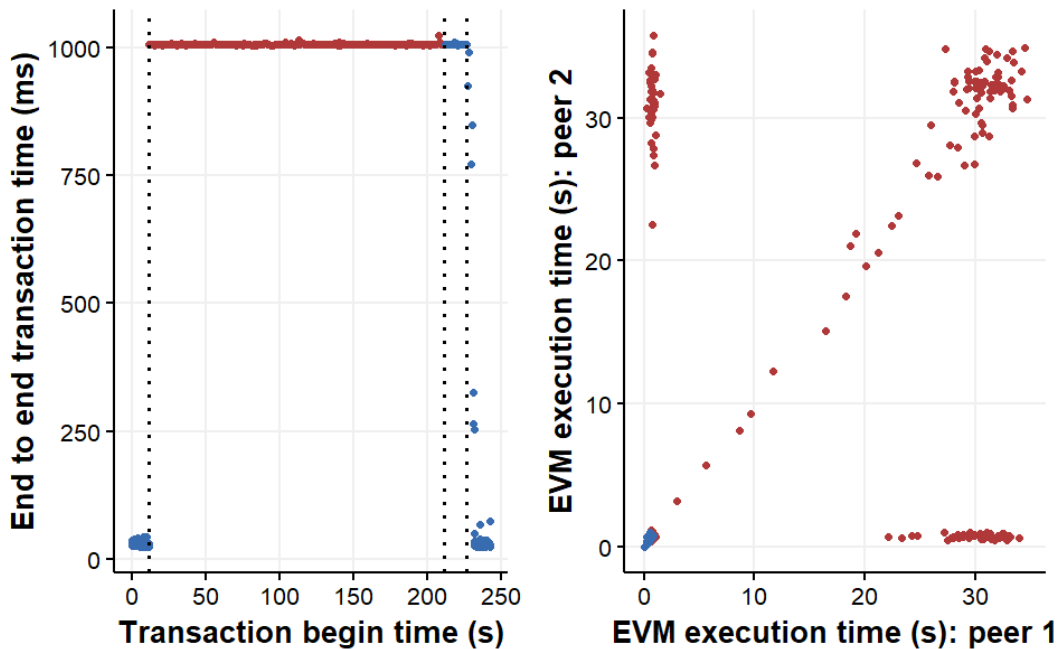
FIGURE 16. End-to-end transaction (attempt) times and Burrow EVM execution times in the timeout campaign (red: faulty, blue: fault-free function).

possibility of smart contract faults having an extra-functional impact on the execution platform and other smart contracts; especially in "business" blockchains that run more complex and comparatively less security and resilience vetted platform software than the public ones.

VI. THREATS TO VALIDITY

In this section, we present the main threats to the validity of this work and discuss mitigation strategies. We start by mentioning that *the number and types of smart contracts used in this work are limited*. We are also aware that libraries for domain-specific smart contract development, such as OpenZeppelin or Hyperledger Grid, are rapidly maturing. Smart contract development is also increasingly utilizing domain-specific languages and generative/interpretative techniques, as available for, e.g., DAML and Hyperledger Concerto. We plan to apply the techniques introduced in this article for these technologies and over a broad set of smart contracts in further research.

Our mutant generation approach can introduce some mutations that lead to code that is strictly behaviorally equivalent to the reference (e.g., swapping two subsequent, data-independent, non-control statements or removing a local overflow check, which is not necessary due to additional global checks). Although it is not affecting the assessment results, filtering these out during mutation generation is ongoing work to avoid duplicate executions.

The defensive mechanisms studies in this work are limited to formal verification and some additional Solidity- and platform-level protection mechanisms, which are the most popular mechanisms. However, further research will target

investigating the effect of introducing further defensive techniques as business-logic derived runtime assertion checks, N-version programming, or, indeed, much of the wide range of fault-tolerant software patterns [62]. These can prove especially important in enterprise blockchain systems, where sizeable, complex smart contracts and critical applications are rapidly appearing.

The manual definition of the fault model may involve some errors due to the human intervention in the process. Thus, some faults may be misclassified, which may affect their overall frequency. To mitigate this threat, the classification was verified by a second researcher, allowing for stronger confidence in the fault model.

Executing one transaction at a time may not be a representative scenario, especially when considering performance metrics. Still, we consider it to be a starting point to execute experiments that may impact performance, and future work may consider scenarios involving the execution of multiple transactions at a time.

The performance analysis of the Blockchain system is limited to the execution time of one faulty transaction compared to its reference run, which is quite relevant. Nevertheless, this study will be extended in the future to include the whole fault model considering other performance attributes.

VII. CONCLUSION

In this article, we presented an approach for assessing key dependability properties of blockchain systems in the presence of faulty smart contracts. The approach is based on injecting a set of software faults that is composed of general cases of software faults and blockchain-specific cases.

We execute runtime tests over the modified faulty contracts to observe divergence from reliable (correct) behavior and ledger state integrity. We also evaluate the effectiveness of formal verification and contract-level defense and protection mechanisms against the injected software faults.

We evaluated a small but – considering the nature of smart contracts – representative set of contracts and found that the combined use of formal verification and protection mechanisms is a powerful way of reducing the number of client observable and residual faults. Still, these techniques are not sufficient to prevent the occurrence of all kinds of faults, with a few cases escaping these verification techniques. We summarize the main findings and observations in the next paragraphs.

Modern blockchain frameworks support the application developer by offering built-in services for the most complex tasks, especially by managing data distribution and synchronization, consensus, and ordering of transactions. Despite this, **current built-in verification tools and services lack adequate support to assure the dependability of such complex systems.** The principle of replicated synchronized ledgers and prior agreement on contracts and their execution by consensus implements an n-out-m modular redundancy scheme providing system-level guarantees for Byzantine or random errors. However, **as software faults in accepted smart contracts corrupt all nodes in a common mode way, replication and consensus are ineffective, and they remain uncovered.**

Solidity and its runtime environments offer limited mechanisms for self-checking and validation of user inputs in the form of assumptions and assertions. **Enriching the contract with assertions and requirements reveals a significant number of logical errors** by blockchain assertion checks. Further **platform level checks detect additional faults** before committing transactions (e.g., timeouts, account validation, balance validation in case of native tokens). **Formal verification can cover a broader subset of faults**, and doing the checks at design time can reduce execution cost by omitting the already proven assertions. In our pilot experiments, latent errors were eliminated, and the number of faults that escape the detection and protection mechanisms in the blockchain system (resulting in client observable failures) was also significantly reduced.

As future work, we aim to focus on the security aspects of blockchain systems. The fault model created and presented in this article includes several faults that can be exploited by attackers leading to severe security issues (e.g., missing *require* on transaction sender (MRTS) or wrong visibility (public) for privateinternal function (PVPF)). In order to properly assess the impact of these kinds of faults/security vulnerabilities, security penetration testing is a required technique that, to the best of our knowledge, has been scarcely explored in the blockchain domain. There is a need for effective tools specialized in security for blockchain systems and that are ready to handle the specificities of these systems. This need is increasingly important, especially considering

the rapid adoption of blockchain systems and their new applications, but is especially difficult due to challenges like the definition of effective attack loads and also the evaluation of the impact of an attack, which must be performed on a distributed, complex system.

This will require to perform a large scale field study to characterize software faults and vulnerabilities as threats to the security of blockchain systems. The goal would be to achieve an extensive characterization of software defects and vulnerabilities in smart contracts across different programming languages. Some initial efforts in this regard exist (e.g., in this work), but none that allows the definition of representative and complete vulnerability models or attack models. In fact, software defects (being them of a security nature or not) need to be detailedly characterized according to various dimensions, type of the defect, type of the impact on the system, necessary conditions that allow the bug to surface (e.g., by using Orthogonal Defect Classification, or ODC+V [48]) and complemented with finer details, such as the typical location or the exact triggering conditions of such type of bug, which tend to be hard to characterize due to the variety of cases (e.g., presence of bugs in recursive functions, bugs that only surface when the system reaches a certain state).

REFERENCES

[1] W. Stallings, "A blockchain tutorial," *Internet Protocol J.*, vol. 20, no. 3, pp. 2–24, Nov. 2017.

[2] D. Puthal, N. Malik, S. P. Mohanty, E. Kougianos, and C. Yang, "The blockchain as a decentralized security framework [future directions]," *IEEE Consum. Electron. Mag.*, vol. 7, no. 2, pp. 18–21, Mar. 2018.

[3] M. O'Dair and Z. Beaven, "The networked record industry: How blockchain technology could transform the record industry," *Strategic Change*, vol. 26, no. 5, pp. 471–480, Sep. 2017.

[4] F. Curbera, D. M. Dias, V. Simonyan, W. A. Yoon, and A. Casella, "Blockchain: An enabler for healthcare and life sciences transformation," *IBM J. Res. Develop.*, vol. 63, nos. 2–3, pp. 8:1–8:9, Mar. 2019.

[5] M. Iansiti and K. R. Lakhani, "The truth about blockchain," *Harvard Bus. Rev.*, vol. 95, no. 1, pp. 118–127, 2017.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.

[7] N. Szabo, "Smart contracts: Building blocks for digital markets," *EXTROPY, J. Transhumanist Thought*, vol. 18, no. 2, p. 2, 1996.

[8] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Gener. Comput. Syst.*, vol. 107, pp. 841–853, Jun. 2020.

[9] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts," in *Principles of Security and Trust* (Lecture Notes in Computer Science), vol. 10204. Cham, Switzerland: Springer, 2017, pp. 164–186.

[10] NIST. (2020). *National Vulnerability Database*. [Online]. Available: https://nvd.nist.gov/

[11] C. Forward and V. Dhillon, "The DAO hacked," in *Blockchain Enabled Applications*, V. Dhillon, D. Metcalf, and M. Hooper, Eds. Cham, Switzerland: Springer, 2017, pp. 67–78.

[12] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[13] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003.

[14] Ethereum. (2019). *Solidity Documentation*. [Online]. Available: https://solidity.readthedocs.io/en/latest/

[15] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[16] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and S. Muralidharan, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, Apr. 2018, p. 30.

[17] Z. Zheng, S. Xie, H. N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *Int. J. Web Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[18] *Hyperledger Burrow*. Accessed: Oct. 19, 2020. [Online]. Available: https://github.com/hyperledger/burrow

[19] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.* Cham, Switzerland: Springer, 2016, pp. 79–94.

[20] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 2–8.

[21] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," 2018, *arXiv:1809.09805*. [Online]. Available: https://arxiv.org/abs/1809.09805

[22] A. Miller, Z. Cai, and S. Jha, "Smart contracts and opportunities for formal methods," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, (Lecture Notes in Computer Science), vol. 11247. Cham, Switzerland: Springer, 2018, pp. 280–299.

[23] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.

[24] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of Ethereum smart contracts," in *Principles of Security and Trust* (Lecture Notes in Computer Science), vol. 10804. Cham, Switzerland: Springer, 2018, pp. 243–269.

[25] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Jan. 2018, pp. 67–82.

[26] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269.

[27] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *CoRR*, vol. abs/1802.06038, pp. 1–15, Mar. 2018.

[28] ConsenSys. (2019). *Mythril Classic: Security Analysis Tool for Ethereum Smart Contracts*. [Online]. Available: https://github.com/ConsenSys/mythril-classic

[29] Trail of Bits. (2019). *Slihter: Static Analyzer for Solidity*. [Online]. Available: https://github.com/trailofbits/slither

[30] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "KEVM: A complete semantics of the Ethereum virtual machine," IDEALS, Tech. Rep., 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8429306

[31] Y. Hirai, "Defining the Ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security* (Lecture Notes in Computer Science), vol. 10323. Cham, Switzerland: Springer, 2017, pp. 520–535.

[32] I. Sergey, A. Kumar, and A. Hobor, "Scilla: A smart contract intermediate-level LAnguage," 2018, *arXiv:1801.00687*. [Online]. Available: http://arxiv.org/abs/1801.00687

[33] B. Beckert, M. Herda, M. Kirsten, and J. Schiffl, "Formal specification and verification of hyperledger fabric chaincode," in *Proc. 3rd Symp. Distrib. Ledger Technol. (SDLT)*, 2018, pp. 1–5.

[34] Z. Nehai and F. Bobot, "Deductive proof of ethereum smart contracts using Why3," 2019, *arXiv:1904.11281*. [Online]. Available: http://arxiv.org/abs/1904.11281

[35] L. Alt and C. Reitwiessner, "SMT-based verification of solidity smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice* (Lecture Notes in Computer Science), vol. 11247. Cham, Switzerland: Springer, 2018, pp. 376–388.

[36] A. Hajdu and D. Jovanović, "Solc-verify: A modular verifier for Solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, 2019.

[37] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain," 2018, *arXiv:1812.08829*. [Online]. Available: http://arxiv.org/abs/1812.08829

[38] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 18–20.

[39] *Truffle Suite*. Accessed: Oct. 19, 2020. [Online]. Available: https://www.trufflesuite.com/

[40] sc-forks. (2017). *Solidity-Coverage-Code Coverage for Solidity Testing*. [Online]. Available: https://github.com/sc-forks/solidity-coverage

[41] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, p. 44:1–44:55, Feb. 2016.

[42] P. Chapman, D. Xu, L. Deng, and Y. Xiong, "Deviant: A mutation testing tool for solidity smart contracts," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Jul. 2019, pp. 319–324.

[43] J. J. Honig, M. H. Everts, and M. Huisman, "Practical mutation testing for smart contracts," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cham, Switzerland: Springer, 2019, pp. 289–303.

[44] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "MuSC: A tool for mutation testing of ethereum smart contract," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1198–1201.

[45] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 259–269.

[46] M. C. R. Ponte and I. Medeiros, "Fuzzing Ethereum smart contracts," in *Proc. DSN Workshop Byzantine Consensus Resilient Blockchains*, 2018.

[47] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 531–548.

[48] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification—A concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[49] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: A call for blockchain software engineering?" in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 19–25.

[50] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories (MSR)*, May 2017, pp. 413–424.

[51] J. Dourlens. (Aug. 2017). *SafeMath to Protect From Overflows*. Accessed: Oct. 19, 2020. [Online]. Available: https://ethereumdev.io/using-safe-math-library-to-prevent-from-overflows/

[52] P. Müller, *Modular Specification Verification Object-Oriented Programs*. Cham, Switzerland: Springer, 2002.

[53] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects* (Lecture Notes in Computer Science), vol. 4111. Cham, Switzerland: Springer, 2006, pp. 364–387.

[54] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Cham, Switzerland: Springer, 2018, pp. 305–343.

[55] *The State Machine Contract*. Accessed: Oct. 19, 2020. [Online]. Available: https://github.com/Azure-Samples/blockchain

[56] *The Wallet Contract*. Accessed: Oct. 19, 2020. [Online]. Available: https://github.com/SRI-CSL/solidity

[57] E. T. S. Anon. *Erc20 Token Standard—The Ethereum Wiki*. Accessed: Oct. 19, 2020. [Online]. Available: https://en.bitcoinwiki.org/wiki/ERC20

[58] CVE-2018-10299: Beauty Ecosystem Coin (BEC), NIST National Vulnerability Database. (2018). *BatchOverflow*. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-10299

[59] *The Escrow Contract*. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts.

[60] *Solidity Compiler*. [Online]. Available: https://github.com/ethereum/solidity

[61] *The Specific Error Codes*. [Online]. Available: https://github.com/aklenik/caliper/blob/ fabric-evm/src/adapters/fabric-ccp/src/ github.com/hyperledger/fabric-chaincode-evm/evmcc/vendor/ github.com/hyperledger/burrow/execution/errors/errors.go#L24

[62] R. S. Hanmer, *Patterns for Fault Tolerant Software*. Hoboken, NJ, USA: Wiley, 2013.

ÁKOS HAJDU is currently pursuing the Ph.D. degree with the Budapest University of Technology and Economics. He was a Visiting Researcher with McGill University and held internships at CERN and SRI International. His research interests include formal methods and logic, with the primary focus on SMT-based software verification. In his field, he has authored more than 20 peer-reviewed articles, participated in national/international research projects, and contributed to various open-source tools.

LÁSZLÓ GÖNCZY received the Ph.D. degree from the Budapest University of Technology and Economics (BME), Hungary. He is currently an Assistant Professor with BME. He has authored more than 40 papers in refereed conferences and journals and participated in several national and international projects. His research interests include system modeling (with an emphasis on business process modeling) and data-driven dependability evaluation, recently in the context of blockchain-based applications.

NAGHMEH IVAKI (Member, IEEE) received the Ph.D. degree from the University of Coimbra, Portugal. She is currently a Postdoctoral Researcher and a Full Member with the Centre for Informatics and Systems (CISUC), Software and Systems Engineering Group (SSE), Department of Informatics Engineering, University of Coimbra. She specializes in the scientific field of informatics engineering, with a particular focus on security and dependability of computer systems. In her field of specialization, she has authored more than 25 peer-reviewed publications and participated in several national and international research projects.

NUNO LARANJEIRO (Member, IEEE) received the Ph.D. degree from the University of Coimbra, Portugal. He is currently an Assistant Professor with the University of Coimbra. His research interests include robustness of software services as well as experimental dependability evaluation, web services interoperability, services security, and enterprise application integration. He has contributed, as the author and a reviewer, to leading conferences and journals in the dependability and services computing areas, and participated in several national and international projects.

IMRE KOCSIS received the Ph.D. degree from the Budapest University of Technology and Economics. He is currently an Assistant Professor with the Budapest University of Technology and Economics. He has authored numerous papers in refereed conferences and journals in the dependability and cloud computing fields, and participated in several national and international projects. His research interests include the resilience assurance of distributed ledger technology-based solutions, and the resilience assurance of cloud and edge systems in a broader context.

HENRIQUE MADEIRA (Member, IEEE) is currently a Full Professor with the University of Coimbra, where he has been involved in research on dependable computing, since 1989. His research interests include software quality and software reliability, experimental evaluation and benchmarking of dependability and security, and fault injection techniques. His current research projects involve two research directions such as assured AI, focusing on providing safety and security guaranties in critical applications that use AI; and human factors in software engineering, particularly on the use of biometrics to improve software quality.

ATTILA KLENIK is currently pursuing the Ph.D. degree in computer science engineering with the Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics. His research interests include benchmarking of distributed ledger technologies, exploratory analysis of performance data, and performability assessment of distributed systems in general.

ANDRÁS PATARICZA graduated in electrical engineering. He received the D.Sc. degree from the Hungarian Academy of Sciences (MTA) and the Dr.Habil. degree from BUTE. In 1994, he founded the Fault-Tolerant Systems Research Group, Department of Measurements and Information Systems. He received multiple recognition awards from different scientific and industrial organizations, like the Academic Award of the MTA, and six IBM Faculty Awards. He has published over 180 articles in international journals, conferences, and workshops in the field of dependable computing, cyber-physical systems, and model-driven engineering. He was the General Chair of 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2013). He serves as the Honorary Chair of the Computer Science and Engineering Branch of the National Council of Student Research.

· · ·