SmartFuzzDriverGen: Smart Contract Fuzzing Automation for Golang

Siddhasagar Pani (https://orcid.org/0000-0002-3771-7860), Tata Consultancy Services, India, siddhasagarpani@gmail.com **(mailto:siddhasagarpani@gmail.com)**
Harshita Vani Nallagonda (https://orcid.org/0000-0003-4304-3767), Tata Consultancy Services, India, harshita.n@tcs.com **(mailto:harshita.n@tcs.com)**
Vigneswaran (https://orcid.org/0000-0002-4013-392X), Tata Consultancy Services, India, vigneswaran.r@tcs.com **(mailto:vigneswaran.r@tcs.com)**
Raveendra Kumar Medicherla (https://orcid.org/0000-0002-9162-4825), Tata Consultancy Services, India, raveendra.kumar@tcs.com **(mailto:raveendra.kumar@tcs.com)**
Rajan M (https://orcid.org/0000-0001-9839-4754), Tata Consultancy Services, India, rajan.ma@tcs.com **(mailto:rajan.ma@tcs.com)**

DOI: https://doi.org/10.1145/3578527.3578538 (https://doi.org/10.1145/3578527.3578538)
ISEC 2023: 16th Innovations in Software Engineering Conference (https://doi.org/10.1145/3578527), Allahabad, India, February 2023

Greybox fuzzers require intermediate programs called *fuzz drivers* to test smart contract APIs. These fuzz drivers use the semi-random inputs (bytes) generated by fuzzers to prepare suitable inputs required to test APIs. Further, fuzz driver also uses this input to decide sequence in which APIs to be invoked and enables the fuzzer to execute the APIs in that sequence to find the vulnerabilities, if any. Manually writing such complex and intelligent fuzz drivers is laborious, requires deep technical skills, hence can be cumbersome and error prone. In this paper, we propose *SmartFuzzDriverGen* framework to automatically generate fuzz drivers which invoke smart contract APIs using different strategies: unit-level, sequence-based (random, user-defined), and heuristics based. We evaluate the proposed framework by testing a prototype implementation of it with Golang smart contracts (targeted for Hyperledger Fabric platform) and study the effectiveness of the generated fuzz drivers in terms of code coverage as well as bug finding abilities. We observed that fuzzing of APIs in random sequences performed better than the other methods.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; • **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**;

Keywords: blockchain, smart contracts, fuzzing, automated driver generation, sequencing, vulnerability detection

ACM Reference Format:
Siddhasagar Pani, Harshita Vani Nallagonda, Vigneswaran, Raveendra Kumar Medicherla, and Rajan M. 2023. SmartFuzzDriverGen: Smart Contract Fuzzing Automation for Golang. In *16th Innovations in Software Engineering Conference (ISEC 2023), February 23–25, 2023, Allahabad, India.* ACM, New York, NY, USA 11 Pages. https://doi.org/10.1145/3578527.3578538 **(https://doi.org/10.1145/3578527.3578538)**

1 INTRODUCTION

Smart contracts are computer programs which represent business logic or agreements that get executed in a blockchain network. Just like any other programs, smart contracts also can have bugs, and there are many incidents reported in the past where adversaries have exploited bugs to steal millions of dollars' worth of tokens [11, 15, 23, 25]. Further, the immutable nature of blockchain, the involvement of multiple stakeholders in the ecosystem and the impacts associated with stopping the operations for patching make it difficult to patch smart contracts after deployment. One approach to address this problem is to use upgradable smart contracts [21]. However, such feature must have been enabled at the time of smart contract development itself. Also, it may introduce new bugs in the due course because of the way smart contract upgrade is realized [10]. Overall, the techniques available for patching smart contracts post deployment are difficult to achieve and error prone. Thus, before deployment, ensuring that a smart contract is devoid of bugs is critical and this can be achieved with high-coverage testing.

Some of the prominent testing methodologies are Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) [12]. SAST techniques look for the known weaknesses in code and report. However, SAST tends to report high number of false positives [22]. On the other hand, fuzzing, a type of DAST technique, reports no false positives. Also, fuzzing is proven to be good at finding unknown vulnerabilities in programs.

Generally grey-box fuzzers (fuzzers) like AFL [31], libfuzzer [17], go-fuzz [28] are unaware of the program under test and are agnostic to fuzzing application program or library. Moreover, testing an application program can be different than that of a library. An application program has a single entry point (such as the main() function) and control flows from the entry point to the other functions based on the inputs and finally reaches the end of the program. On the other hand, a library has a collection of independent APIs and every API can be an entry point and control does not flow automatically between the APIs. Hence in order to test all the APIs of a library, fuzzers require some intermediate program (fuzz driver) to mimic a generalized end-user of the library which invokes the APIs with appropriate inputs and in required or intended sequences. The fuzz driver parses the semi-random input generated by the fuzzer into API input parameters and invokes the APIs. As smart contracts can be viewed as libraries having a collection of APIs which are stateful, fuzz drivers are needed for testing smart contracts also.

To test stateful APIs and reach maximum code coverage, fuzz driver is required to call the APIs in different sequences. Also, some bugs can be triggered only when the function calls are made out of sequence (unexpected or invalid sequence). With *valid sequence testing*, correctness of expected behaviour of the software is tested. Whereas, with *invalid sequence testing*, potential vulnerabilities present in the invalid paths can be identified. Invalid sequence can be treated as *negative testing*, to imitate exploiter's intention. Thus fuzz drivers carry significant intelligence in making the fuzz testing efficient. When the smart contract under test is complex with several APIs and multiple valid and invalid sequences can exist, the complexity of the driver increases. Hence, listing the valid or invalid sequences exhaustively can be infeasible. To use continuous integration, continuous delivery (CI/CD) pipeline for smart contract development, automation of fuzz driver generation is crucial as it requires the driver to be updated whenever a new API is added to a smart contract or there is a change in existing API parameters. Also, manually writing such fuzz drivers is cumbersome, laborious, requires deep technical skills, hence can be error prone.

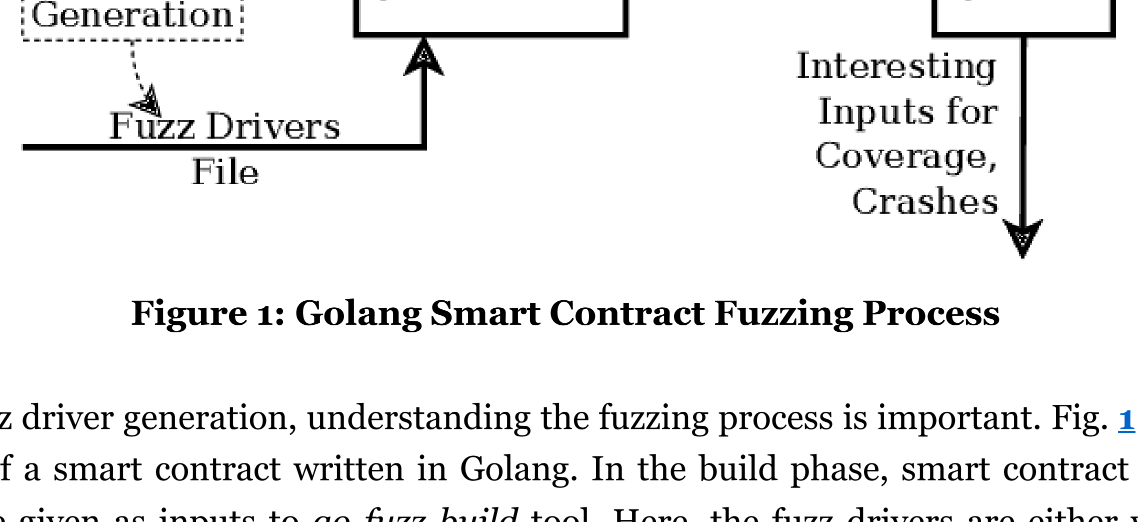


Figure 1: Golang Smart Contract Fuzzing Process

To automate the fuzz driver generation, understanding the fuzzing process is important. Fig. 1 shows the overview of fuzzing process of a smart contract written in Golang. In the build phase, smart contract files as well as fuzz drivers (*fuzz.go*) are given as inputs to *go-fuzz-build* tool. Here, the fuzz drivers are either written manually or generated using an automated method as shown in dotted lines. This phase instruments the smart contract code, builds along with drivers, and creates a.zip file. In the test phase, the.zip file that was generated in the build phase is taken as input and the fuzzer *go-fuzz* starts the fuzz testing in an infinite loop. As part of this process the fuzzer identifies inputs which cause new code coverage or new crashes in the smart contract and store them in the directories *corpus* and *crashers* respectively.

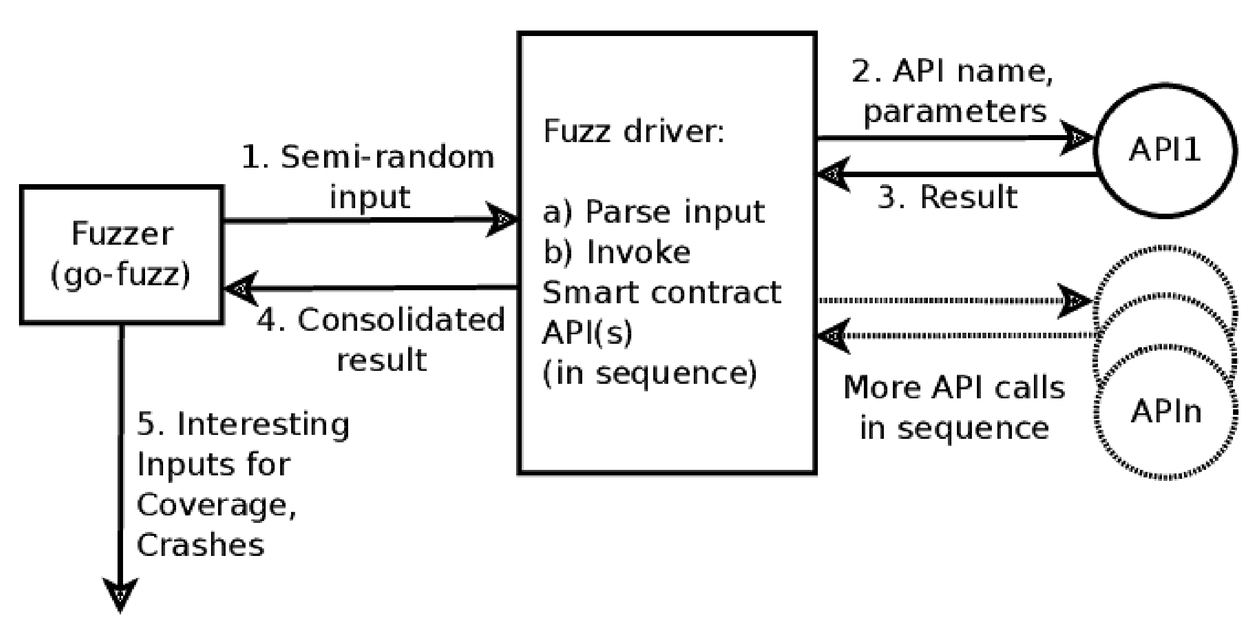


Figure 2: Fuzz Test Control Flow for One Cycle

Fig. 2 shows how the control flows between different components while fuzzing (for a fuzz cycle or for one input generated by the fuzzer). Fuzzer provides a semi-random input, byte array, to the fuzz driver. The fuzz driver decides the next API to be invoked (based on the algorithm discussed in section 4), parses the input into suitable parameters for the API, invokes the API (and if necessary, more than one APIs in a sequence), and returns the invocation result to the fuzzer. Fuzzer outputs the interesting inputs as mentioned earlier.

In this work, we propose *SmartFuzzDriverGen*, a framework to automatically generate a set of fuzz drivers and we demonstrate our framework by instantiating it for Hyperledger Fabric smart contracts written in Golang. These fuzz drivers execute smart contract APIs in different sequences (with and without preserving the states across the API calls) to increase coverage and in turn to find bugs efficiently. For example, calling APIs in a) random order (with and without allowing calling of an API repeatedly), b) heuristics-based order (such as sorting the APIs based on number of lines of code present in them or based on complexity of the APIs in terms of number of nested conditions/loops), and c) user-defined order. Essentially, these are chosen to cover positive as well as negative testing scenarios, and the rationale behind choosing each order is explained in section 4. Further, we compare the results to get insights on which strategy performs better in terms of code coverage and finding bugs with fuzzing.

The rest of the paper is organized as follows: We provide a motivating example in Section 2 and cover the required preliminaries in Section 3. We present our approach in Section 4. This is followed by results and discussions of the experiments mentioned in Section 5. We present related work in Section 6. Finally, conclusions and future work are discussed in Section 7.

2 MOTIVATING EXAMPLE

We use a smart contract corresponding to a simple supplychain workflow, as our motivating as well as running example.

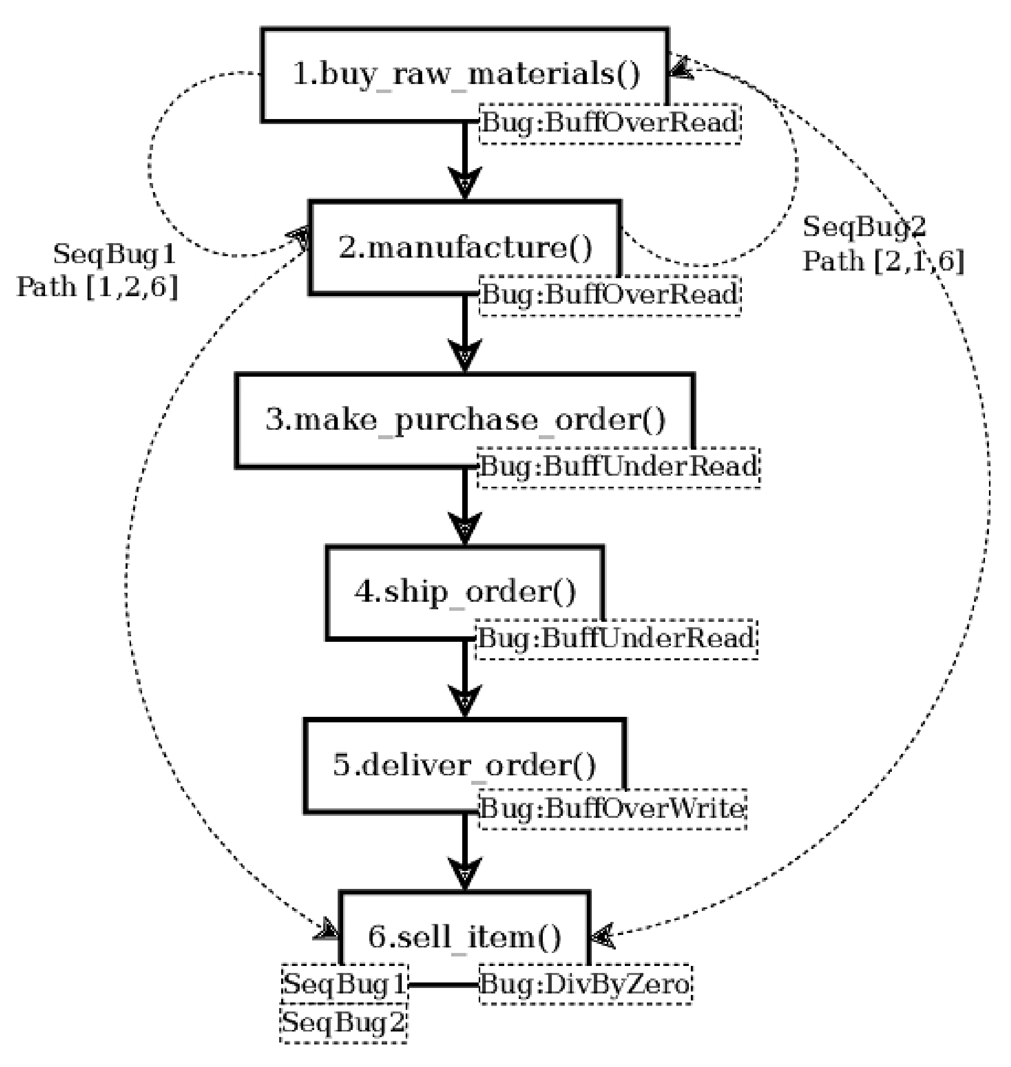


Figure 3: Motivating Example: Supplychain Workflow

The motivating example shown in Fig. 3 has the APIs *buy_raw_materials()* to procure raw materials, *manufacture()* to tag the manufactured goods, *make_purchase_order()* to create a purchase order for the manufactured goods, *ship_order()* to change the state of the goods once shipped, *deliver_order()* to change the state of the goods once delivered, *sell_item()* to change the state of the goods once sold. These are meant to be called by different participants of the supplychain eco system, such as manufacturers, distributors, logistics firms, and retailers. The workflow in Fig. 3 depicts a valid sequence of API calls to be made (see solid arrows). The functionalities of these APIs are not significant for our purposes, however the workflow is. In the example we manifested bugs in different APIs in such a way that some of them will be triggered while calling the APIs independently and some of them will be triggered while calling the APIs in certain sequences. The bugs are mentioned inside the smaller rectangles (with dotted border). The independent bugs are named as *"Bug:* ⟨*bugtype*⟩*"* and the sequence bugs are named as *"SeqBug*⟨*number*⟩*"*. The paths to reach the sequence bugs are shown as dotted lines. Sequence bugs can be triggered only when the APIs are called in the appropriate sequence (and if some conditions are met). For example, SeqBug1 gets triggered when the APIs {1,2,6}, that is, [*buy_raw_materials()*, *manufacture()*, *sell_item()*], are called in that order. This is part of a valid sequence of API calls. On the other hand, SeqBug2 gets triggered when the APIs {2,1,6}, that is, [*manufacture()*, *buy_raw_materials()*, *sell_item()*], are called in that order. This is an invalid or unexpected sequence of API calls. *SmartFuzzDriverGen* can consider both of these scenarios into account while generating fuzz drivers. More information on the bugs and fuzz driver generation approach are discussed in the following sections.

3 BACKGROUND

In this section, we present a brief introduction to the topics like Hyperledger Fabric, Smart Contracts with their types, and Fuzzing that are relevant to understand our proposed work in the paper.

3.1 Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain platform which supports writing of smart contracts in programming languages like GO, Java, and NodeJS. This is an account based blockchain where the ledger has two parts; 1. immutable, append-only blockchain, 2. mutable *world state* [9]. The former contains the list of transactions submitted so far and the latter contains the current values or cumulative results of executing the transactions, as key value pairs. If we consider blockchain entries as our bank transactions, then world state holds/represents the current balance. HLF provides APIs for smart contracts to store values into world state using the corresponding keys. These ledger states are specific to smart contracts and are persistent across smart contract API calls. Thus smart contract APIs are stateful.

We use the terminology state to mean any state variable stored on the ledger.

3.2 Smart Contracts

Smart contracts are the programs that implement business logic. The execution results of smart contract APIs are stored immutably in blockchain which become a tamper proof record of events or transactions. This self-enforcement of rules avoids the need for intermediaries which were needed otherwise to bring trust in executing transactions in various businesses.

HLF's smart contract specifications have been evolved over time. Based on the way HLF expects the smart contracts to be written, we classify them into two major types *Type1* and *Type2*. The main difference between these types is whether API parameters can be obtained from the API signature itself or not.

3.2.1 Type1. These are smart contracts which implement two public functions named *Init* and *Invoke*. Other APIs in the smart contract are implemented as different functions (which need not be public) and called from one of *Init* or *Invoke* function. The smart contract APIs which are called from *Init* are meant for initializing the states of the smart contract and the ones called from *Invoke* are meant for implementing the functionalities of the smart contract. To call a particular API, end-user passes the API name and a set of inputs as a string array to *Init*/*Invoke* (as per HLF's specification).

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
 function, args := stub.GetFunctionAndParameters()
  [...]
} else if function == "buy_raw_materials" {
 return t.buy_raw_materials(stub, args)
  [...]
```

The above code shows how *Invoke* calls the *buy_raw_materials* API based on the function name passed by the end-user.

```
func (t *SimpleChaincode) buy_raw_materials (stub shim.ChaincodeStubInterface, args []string) pb.Response
```

The above code shows the signature of the API *buy_raw_materials* from our motivating example (written in Golang), in *Type1* implementation. This API is a member function of *SimpleChaincode* structure. It expects the first parameter to be of type *shim.ChaincodeStubInterface* which will be provided by the HLF platform. The second parameter is a *string array* through which end-user inputs are passed to the API. It returns a value of type *pb.Response*.

3.2.2 Type2. These are smart contracts in which APIs are defined as public functions and receive end-user inputs in independent parameters. Unlike *Type1* where *Init* and *Invoke* are the only public functions or entry points, here all APIs are entry points and can be directly called by the end-users. In Golang, the functions whose name start with a capital letter are considered as public functions which can be called from other programs.

```
func (s *SmartContract) BuyRawMaterials (ctx contractapi.TransactionContextInterface, invoice_id string, materials_details string) error
```

The above code shows the signature of the API *BuyRawMaterials* from our motivating example, in *Type2* implementation. It expects the first parameter to be of type *contractapi.TransactionContextInterface* which will be provided by the HLF platform. The remaining parameters *invoice_id* and *materials_details* are end-user inputs to the API. It returns a value of type *error*. In Type2, the number of input parameters and their types can be identified from the API signature itself.

HLF Testing Framework: In order to execute smart contracts without deploying into a blockchain network, HLF provides libraries such as *shim*, *shimtest*. We are making use of them in the generated fuzz drivers to invoke smart contract APIs while fuzzing.

3.3 Fuzzing

Fuzzing is a testing technique where program under test is executed with multiple random inputs to find out vulnerabilities which may result in unexpected behaviour, memory leaks and crashes. Following tools are used in our experiments:

3.3.1 go-fuzz. This is a coverage guided fuzzer for programs written in Golang [28]. Its fuzzing logic is very much based on American Fuzzy Lop (AFL), a greybox fuzzer. This tool is used in our testing to execute smart contracts with random inputs.

3.3.2 gofuzz. This is a library for populating GO objects with random values [16]. This is used to convert random data generated by go-fuzz into API parameters of different types.

3.4 Key Challenges to Driver Generation

The purpose of a fuzz driver is to make fuzz testing of smart contracts more efficient. Hence, it is important for a fuzz driver to generate wide range of (valid/invalid) inputs for APIs and arrive at API calling sequences. Following are some of the challenges in achieving them while generating fuzz drivers automatically.

- Identifying the type of a smart contract (as *Type1* or *Type2*) is tricky as blockchain smart contracts development framework is evolving over time and there are no coding standards for smart contracts at present. Hence, developers can write smart contracts on their own way or style and also smart contracts can differ based on the version of the platform APIs and interfaces used.
- Identifying the number and type of API inputs from the API signature/declaration is a challenge for *Type1* smart contracts. Also, mapping fuzzer input to API parameters precisely is difficult. Because, *Type1* smart contract APIs take end-user inputs as a single string array which doesn't reveal the number of inputs and their types, as described in section 3.2.1.
- Automatically deriving the valid as well as invalid sequences of API invocations is hard for both type of smart contracts from the smart contract itself, unless the functional specifications are known.
- Similarly, it is hard to identify the data flow dependencies between APIs from the HLF smart contracts as they don't use global variables to store state information. HLF smart contracts store data as key-value pairs in local variables and use platform APIs *GetState* and *PutState* to read data from and write data persistently into world state, respectively. Hence, identifying which key is being read or written is a challenge.

In the following sections we discuss how the above challenges are handled in different components of *SmartFuzzDriverGen* framework.

4 OUR DESIGN APPROACH

In this section, we describe the components and functionalities of the proposed framework *SmartFuzzDriverGen* that addresses the challenges outlined in section 3.4. Our framework utilizes a predefined set of templates and metadata from the developer to generate effective fuzz drivers.

We consider two types of users in the proposed work. (i) *end-user* who supplies parameters to smart contracts, (ii) *developer* who specifies parameters to the fuzz driver generation. Developer can influence the driver generation by providing a set of valid sequences of API calls in the form of a graph as discussed in section 4.3.

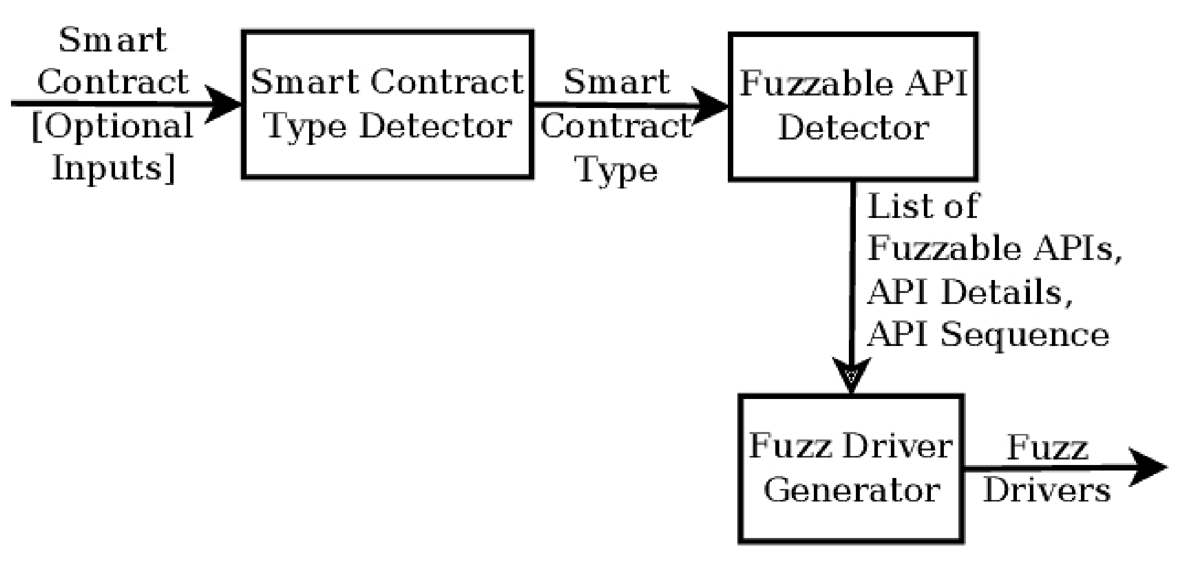

Smart Contract [Optional Inputs]
Smart Contract Type Detector
Smart Contract Type
Fuzzable API Detector
List of Fuzzable APIs, API Details, API Sequence
Fuzz Driver Generator
Fuzz Drivers

Figure 4: *SmartFuzzDriverGen* Components

Fig. 4 shows the components of our framework, 1. Smart Contract Type Detector, 2. Fuzzable API Detector, and 3. Fuzz Driver Generator. Each component is an API to be implemented by the platform/language specific prototypes. The functionalities of each component are briefly described in the following subsections and the implementation details specific to HLF smart contracts written in Golang are given which can be extended to other blockchain platforms and smart contract languages.

4.1 Smart Contract Type Detector

This is the first component of our framework and it identifies the type of smart contract based on criteria such as API prototype/signature/version used in the smart contract, the imported platform API package/path.

For HLF Go lang smart contracts, importing of the package *shim* from either of the following paths corresponds to *Type1* smart contract.

```
github.com/hyperledger/fabric-chaincode-go/shim
github.com/hyperledger/fabric/core/chaincode/shim
```

Similarly, importing of the package *contractapi* from the following path corresponds to *Type2* smart contract.

```
github.com/hyperledger/fabric-contract-api-go/contractapi
```

4.2 Fuzzable API Detector

This is the second component, and it builds an abstract syntax tree (AST) from the smart contract files. Then it traverses the AST to identify the list of fuzzable APIs and some metadata such as length or complexity of those APIs which are needed for the heuristics based fuzz driver generation as described in Section 4.3.3.

The following subsections describe the types of HLF Golang smart contracts, the methods to identify fuzzable APIs in each type, and the achievable level of automation for each type of smart contracts.

4.2.1 Semi-automation for Type1 smartcontracts. In *Type1* smart contracts, Fuzzable APIs are the functions which are called from the public functions *Init* and *Invoke* identifies these APIs from the AST and these APIs can be identified from the AST. However, since the end-user inputs are inside a string array the number of API inputs and their types cannot be determined as mentioned earlier in Section 3.4. Hence, full automation is hard to achieve; the API inputs and their types are taken from the developer as a specification. This JSON has the following elements,

(1) API name: Name of the API is used as the key in the top level

(2) *args*: List of arguments and their types as key-value pairs

(3) *apinametobepassed*: An optional flag which tells whether fuzzable API name need to be passed as part of the input or not. When multiple APIs can be called from Init or Invoke, the actual API name needs to be passed and hence the value of this variable will become 1. Otherwise, 0. This flag needs to be passed only when *SmartFuzzDriverGen* can not identify this information automatically from a smart contract.

(4) *init_invoke_flag*: A flag which tells whether fuzzable API should be called from *Init* function or *Invoke* function. This information helps the fuzz driver to invoke the API appropriately. For APIs which are called from Init function, this flag will have value 0. Otherwise, 1.

Following code shows the format of the specification.

```
"buy_raw_materials": {"args": {"invoice_id": "string", "materials_details" : "string"}, "apinametobepassed": 1, "init_invoke_flag": 1}
```

4.2.2 Full-automation for Type2 smartcontracts. In case of *Type2* smart contracts, the API names, parameters and their types can be identified from the AST itself as mentioned in Section 3.2.2. Thus no additional inputs are required from the developer and full-automation is achieved.

4.3 Fuzz Driver Generator

This is the third component and it generates the following types of fuzz drivers involving all the identified fuzzable APIs based on predefined templates,

(1) Unit level drivers

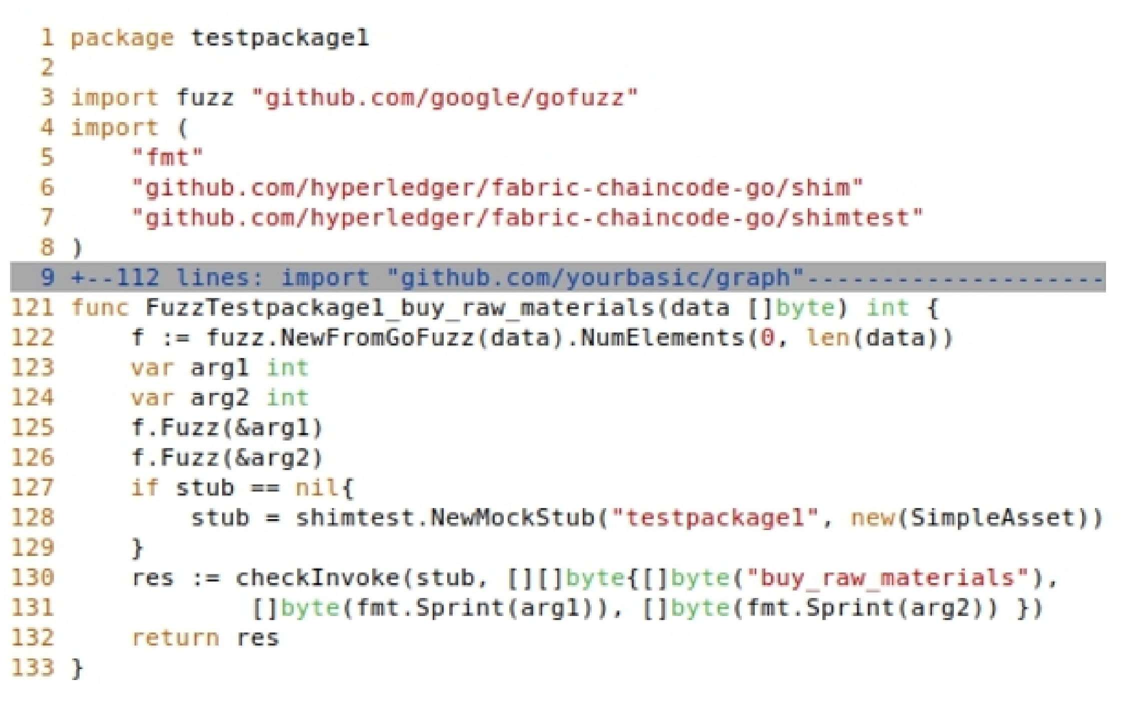
(2) Sequence based drivers (random, user-defined)

(3) Heuristics based drivers

Following subsections describe these drivers in detail.

4.3.1 Unit Level. These drivers invoke smart contract APIs individually without persisting any states across invocations.

Fig. 5 shows a unit level driver for *buy_raw_materials()* API generated by a prototype implementation of *SmartFuzzDriverGen* for HLF Golang smart contract. The driver *FuzzTestpackage1_buy_raw_materials()* receives a byte array *data* as input from the fuzzer and returns an integer. The driver uses *data* as the seed and generates two integers *arg1* and *arg2* using *gofuzz* tool. Then it invokes the smart contract API *buy_raw_materials()* by passing *arg1* and *arg2* through HLF's smart contract testing infrastructure *shimtest* (via the supporting function *checkInvoke()*). Finally, the driver returns the return value of the smart contract API invocation, to the fuzzer. The return value is used by the fuzzer as one of the factors to decide whether the generated input is an interesting input (and to be preserved) or not.



```go
package testpackage1

import fuzz "github.com/google/gofuzz"
import (
    "fmt"
    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-chaincode-go/shimtest"
)
+--112 lines: import "github.com/yourbasic/graph"--------------------
func FuzzTestpackage1_buy_raw_materials(data []byte) int {
    f := fuzz.NewFromGoFuzz(data).NumElements(0, len(data))
    var arg1 int
    var arg2 int
    f.Fuzz(&arg1)
    f.Fuzz(&arg2)
    if stub == nil{
        stub = shimtest.NewMockStub("testpackage1", new(SimpleAsset))
    }
    res := checkInvoke(stub, [][]byte{[]byte("buy_raw_materials"),
            []byte(fmt.Sprint(arg1)), []byte(fmt.Sprint(arg2)) })
    return res
}
```

Figure 5: Sample Fuzz Driver in *fuzz.go* (partial)

4.3.2 Sequence Based. As we discussed earlier with the motivating example, smart contract APIs are stateful and it is important to persist state across smart contract API calls to simulate real world (intended and unintended) usage of APIs in valid and invalid sequences. Invalid sequences of API call can occur when the conditions which safeguard from invalid sequences are weak and can be exploited by malicious users or attackers. Our motive is to find bugs in both valid and invalid paths. Hence, our framework generates drivers to call smart contract APIs in a random order and in the user-defined order as explained below.

Random Sequence Based. These drivers invoke smart contract APIs in a random order, while persisting states across API calls.

Algorithm 1: Random Fuzz Driver Generation

```
Input: Fuzzable_API_List, API_Details
Output: Target file containing Fuzz drivers
Function Create_Random_Seq_Driver():
    Create API_Sequence_With_Out_Repetition
      using Fisher-Yates shuffle algorithm
    Create API_Sequence_With_Repetition by
      choosing APIs randomly from
      Fuzzable_API_List for 'n' times where n =
      number of APIs in Fuzzable_API_List
    API_Sequence = (randomness_condition == 0)
      ? API_Sequence_WO_Repetition :
      API_Sequence_With_Repetition
    for API ∈ API_Sequence do
        Invoke unit level fuzz driver for this API
          while maintaining state across the
          invocations
    end
    return random driver
end
```

Algorithm 1 describes the random sequence fuzz driver generation steps of our prototype implementation, in an abstract manner. It takes the list of fuzzable APIs and their parameters with types and generates random-sequence driver. The generated driver decides whether to create a random sequence of API calls with repetition or without repetition in a random fashion at the time of execution with fuzzer. When repetition is not allowed, it uses *Fisher-Yates shuffle* algorithm [29] to produce a random permutation in an unbiased manner. When repetition is allowed it randomly selects APIs from the list without any restrictions. In both cases, the length of the sequence is configurable and randomness used in decision making are based on fuzzer inputs; hence, they are reproducible.

User-defined Sequence Based. These drivers invoke smart contract APIs in a user-defined order while persisting states across API calls. Note that in some combinations, random sequence driver can populate valid sequences of API calls. However, if the number of API calls are very large, probability of generating valid sequences of API calls can be very low, and it may take a while for the random algorithm to populate such sequences. Thus in order to speed up the process these drivers are generated by taking valid or expected API sequences from the developer in the form of a graph as shown in Fig. 6. Developer may derive the valid sequences based on his/her prior knowledge about the workflow, from test cases, log traces, or client applications when available.

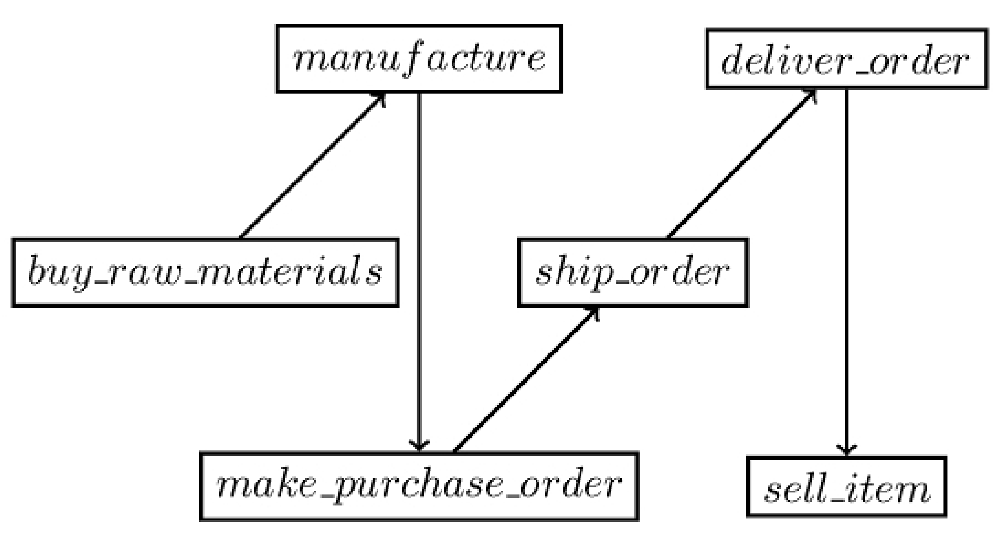


Figure 6: User-defined Sequence Graph

Though in practical scenarios there can be loops in the sequences, the example that we have considered for this work resulted in a loop-free automata.

4.3.3 Heuristics Based. These drivers call smart contract APIs based on some heuristics which can be decided based on the characteristics of the smart contracts. We can fuzz the smart contract APIs by sorting them based on measures such as complexity of the code (number of nested conditions/loops, array operations etc.) and call the API with highest complexity first and then the API with next highest complexity and so on.

In our prototype implementation, we used lines of code as the complexity measure and called longest API first to cover the large functionality initially and then the second longest API and so on.

5 IMPLEMENTATION AND EVALUATION

We evaluated our framework *SmartFuzzDriverGen* with the prototype implemented for HLF smart contracts written in Golang by conducting a set of experiments. Details of the implementation, experiments, and the corresponding observations are discussed in this section. With the help of these experiments we answer the following research questions,

- **RQ1**: Are the generated fuzz drivers syntactically and semantically correct?
- **RQ2**: How did the drivers perform?
- **RQ3**: Which driver performed best?

5.1 Implementation

We conducted the experiments on a 2.1GHz AMD Ryzen 5 3500u machine (8 cores) with 6GB RAM running Ubuntu 20.04.03 LTS. Different tools used in the setup and their versions are as follows:

- go-fuzz (SHA1ID of git commit level): fca39067bc7270ea0dd1a7ce4443eba66ff58fe
- gofuzz: v1.2.0

We used Python and GO to implement different components of *SmartFuzzDriverGen*. The code which does parsing of smart contracts into AST is in GO. It uses the packages *ast*, *parser*, *token* to parse and traverse through the code blocks. The main program is written in Python which makes use of the parser component written in GO to get the necessary details about the smart contract under test. The fuzz drivers generated by *SmartFuzzDriverGen* are in GO. Here, the package *google/gofuzz* is used to convert fuzzer generated inputs into the API parameters. The package *yourbasic/graph* is used to build graph from the developer input on valid API sequences and traverse. Besides, we used unit-testing infrastructure, APIs provided by the HLF platform to implement the complete prototype.

5.2 Benchmarks

As part of benchmarking we conducted two types of experiments. The first experiment is to ensure the syntactic and semantic correctness of the drivers generated. Here, we used smartcontracts taken from *fabric-samples* repository [8], provided by the Hyperledger Fabric community. The second experiment is to understand how different fuzz drivers perform. Here, we used a synthesised smart contract with different bug instances due to the scarcity of benchmark smart contract code in the public domain for HLF when compared to Ethereum blockchain.

5.2.1 Correctness Experiment. Smart contracts used for doing correctness test are listed in Table 1. The objective of this experiment is to ensure the correctness of the drivers generated by our framework by testing with multiple smart contracts and to answer RQ1. In this experiment, we run *SmartFuzzDriverGen* on a set of smart contracts to generate a set of fuzz drivers for each of them. Then we run *go-fuzz-build* and *go-fuzz* commands to build and run fuzz testing of the smart contracts using each of the generated fuzz drivers (they are, unit level, random sequence, user-defined sequence, heuristics based) for 10 seconds each. We consider the result of the correctness test as *successful* if there are no errors reported in the generation, build, and execution phases of fuzzing. Table 1 lists the smart contracts chosen for this experiment. These were taken from *fabric-samples* repository [8], provided by the Hyperledger Fabric community.

Table 1: Smart contract suite used for correctness evaluation

s. no.	smart contract name	lines of code	type	fabric-samples version	fuzz driver status		
					generation	build	test
1	sacc-old	97	1	release-1.0	✓	✓	✓
2	marbles-old	627	1	release-1.0	✓	✓	✓

3	fabcar-old	204	1	release-1.0	✓	✓	✓
4	balance-transfer-old	203	1	release-1.0	✓	✓	✓
5	sacc	97	1	v2.3.0	✓	✓	✓
6	marbles	755	1	v2.3.0	✓	✓	✓
7	interest_rate_swap	313	1	v2.3.0	✓	✓	✓
8	high-throughput	396	1	v2.3.0	✓	✓	✓
9	asset-transfer-abac	214	2	v2.3.0	✓	✓	✓
10	asset-transfer-ledger-queries	466	2	v2.3.0	✓	✓	✓
11	asset-transfer-private-data	780	2	v2.3.0	✓	✓	✓
12	asset-transfer-secured-agreement	732	2	v2.3.0	✓	✓	✓
13	marbles_private	556	2	v2.3.0	✓	✓	✓
14	abstore	135	2	v2.3.0	✓	✓	✓
15	fabcar	151	2	v2.3.0	✓	✓	✓
16	auction	711	2	v2.3.0	✓	✓	✓
17	commercial-paper	650	2	v2.3.0	✓	✗	-
18	token-erc-20	497	2	v2.3.0	✓	✓	✓
19	token-utxo	221	2	v2.3.0	✓	✓	✓

The table lists the name of the smart contracts, lines of code in them, their types (*Type1* or *Type2*), the *fabric-samples* repository version under which the smart contract source code is available, and the fuzz driver status (whether the generation, build, and test phases succeeded or not). Syntactic correctness means that the generated drivers are free of syntax errors. A successful build using *go-fuzz-build* of a driver without any errors ensures that the driver is syntactically correct. Semantic correctness means the generation of drivers for all the fuzzable APIs in a smart contract for all the test strategies. This can be ensured by verifying the drivers manually.

5.2.2 Effectiveness Experiment. Publicly available smart contracts that we used in our correctness experiments are simple and meant only for illustrating different features of HLF; thus, we could not find any significant bugs in them while fuzzing. Hence, we synthesised a smart contract with bugs which is shown in our motivating example, for testing effectiveness of automatic fuzz driver generation framework.

The motivation behind introducing bugs in certain manner was inspired from [26]. The authors describe certain properties for bugs which are said to be in hard-to-find program locations. Five of them which we used are listed below,

label=() *Reproducible*: where the bug triggering inputs can be reproduced any number of times in real time

lbbel=() *Deep*: so that bugs are at considerable depth so as be made hard to be covered and here it is with-respect-to the complexity of the general smart contract code

lcbel=() *Rare*: where the guard or predicate conditions make the bug to be triggered on very few generated inputs only

ldbel=() *Simple*: guard conditions do not change the actual control flow of the contract

lebel=() *Non-trivial* and *Useful*: least number of inputs trigger them

The synthesised smart contract has six well-known bugs for our experimentation (as shown in Fig. 3). They are Divide-by-zero (CWE-369), Buffer-underflow (both index and slice) (CWE-127), Buffer-overflow (index and slice) (CWE-126) and Buffer-overwrite (CWE-787). We have guarded the bugs with conditions in such a way that five of the bugs are relatively easier to trigger and one of them (buffer-overwrite) is guarded with a relatively more stringent condition. Then we introduced two more bugs, using *panic(nil)* which would be triggered only if a set of APIs are invoked in a certain sequence as well as certain conditions on the API inputs are satisfied. We picked three APIs through which guards were introduced as predicates so that, if and only if those predicates were satisfied the bugs would be triggered. This is represented as a state machine in Fig. 7.

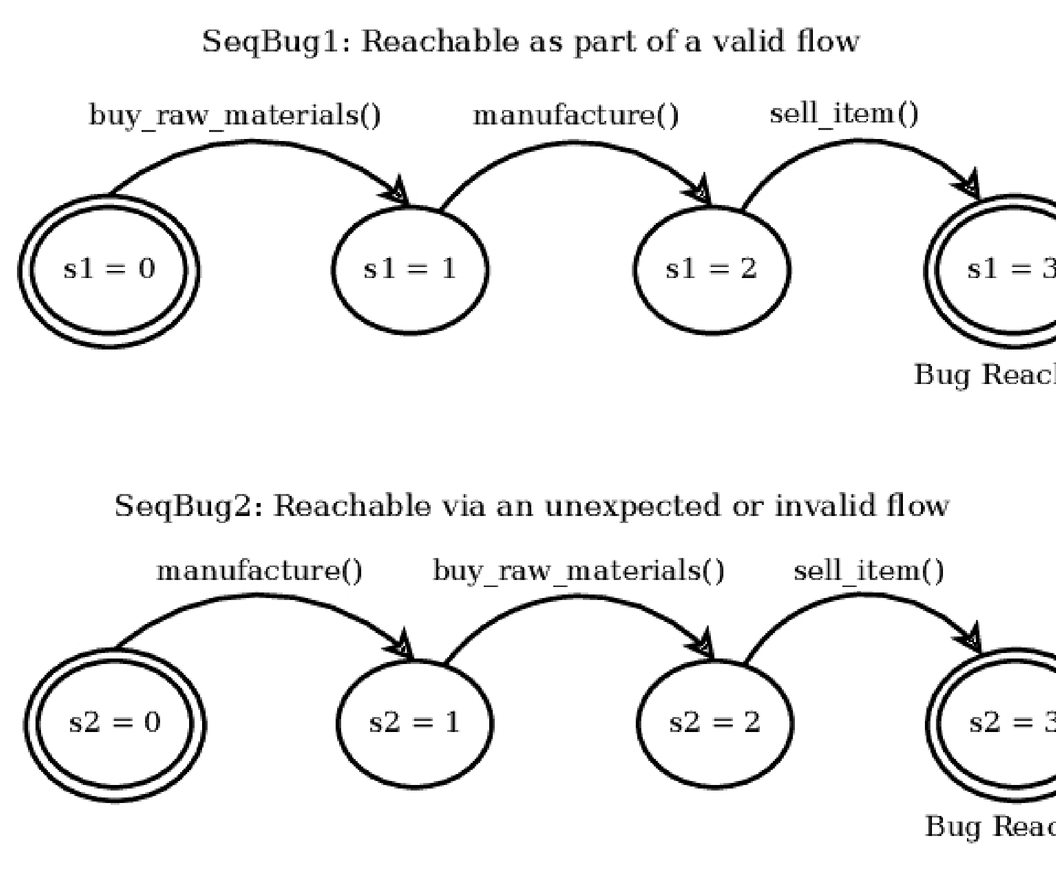


Figure 7: State machine representing API calling sequences to reach sequence bugs

State variables *s1* and *s2* are used to track the state changes towards reaching the sequence bugs *SeqBug1* and *SeqBug2* respectively. The API sequence *buy_raw_materials()* → *manufacture()* → *sell_item()* triggers *SeqBug1*. This is part of a valid work flow. The API sequence *manufacture()* → *buy_raw_materials()* → *sell_item()* triggers *SeqBug2*. This is part of an invalid or unexpected work flow. The experiment was carried out 5 times for each of the strategy as 1 hour runs each.

5.3 Correctness Evaluation

Results of this experiment are manually verified by authors by using *-testoutput* option while running go-fuzz to study the outcome of the fuzzing runs. The following observations answers the research question **RQ1** about ***correctness*** of the drivers.

- *SmartFuzzDriverGen* is able to generate fuzz drivers for all smart contracts (except for one as explained later in this section).
- The generated fuzz drivers are syntactically and semantically correct in all these cases and the build (with *go-fuzz-build*) and test (with *go-fuzz*) went fine without any errors from the driver side.

Observed exception in generating fuzz driver and the reason are given below,

- From Table 1 we can observe that fuzz driver generation happened successfully for the smart contract *17.commercial_paper*. However, the driver failed in the build phase. This use case consists of two *Type2* smart contracts. *Type2* smart contract APIs receive the first parameter of type *contractapi.TransactionContextInterface* which is automatically provided by the blockchain platform (not taken from the end-user). Hence, the generated fuzz driver prepares and passes such parameter to the API via HLF's testing interface. However, APIs in the *commercial_paper* smart contracts take a customized *TransactionContextInterface* type input as the first parameter. This custom type has an additional method named *GetPaperList()* and hence the error *missing method GetPaperList* in the build phase.

5.4 Effectiveness Evaluation

The objective of this experiment is to benchmark performance of different types of drivers generated for APIs of a particular smart contract using different strategies. Then answer RQ2 on the ability of these drivers in terms of code coverage as well as bugs identified.

Table 2: Bug and Coverage with various drivers

Best of 5 test runs, with 1hr each									
Type of execution	**Coverage(%)**	**First Identification Time (in seconds)**							
		BORI	BORS	BURI	BURS	OOBW	DBZ	SeqBug1	SeqBug2
unit level	75.5	798.08	1088.07	263.94	739.27	-	59.16	NA	NA
random order	80	1224.27	1194.49	979.25	1132.86	-	173.25	1289.17	1288.75
user-defined order	74.32	862.51	875.98	393.37	625.52	-	57.33	1116.62	NA
heuristics (no. of lines)	75.5	318.82	470.96	141.58	470.56	-	76.94	NA	NA

BORI: Buffer Over Read Index; BORS: Buffer Over Read Slice; BURI: Buffer Under Read Index; BURS: Buffer Under Read Slice; OOBW: Out Of Bounds Write; DBZ: Divide By Zero; SeqBug1: Sequence Bug1; SeqBug2: Sequence Bug2

In Table 2, we put forth the values of the first incident time of the various bugs and percentage of code covered. The experiment was carried out 5 times for each of the strategy as 1 hour runs each. The values entered in the table are the geometric mean of the values from the five runs. Unit level and heuristics based tests are not meant for identifying sequence bugs. Hence, we mentioned "NA" in the "First Identification Time" column corresponding to those tests. Conversely, we use "-" to denote missing of certain bugs within the experiment duration by the tests which are meant to identify those bugs.

Fuzz driver generated by *SmartFuzzDriverGen* identified unexpected bugs which were not found earlier from two different smart contracts from our internal repository,

- JSON object de-serialization call caused a crash when the fuzzer generated UTF-32 characters as part of the input.
- 'nil' pointer dereference errors were reported due to uninitialized objects. These are expected to be initialized by the blockchain platform APIs.

5.4.1 Discussion on Coverage. We collected the corpus generated from all the experimental runs and chose one of them which is very similar or close to the geometric mean results entered in Table 2. Then we ran the corpus of inputs in unit tests and used Golang's native cover tool to get the coverage percentage against the inputs that were generated up to a particular time and constructed a graph for all the different strategies. Fig. 8 depicts the coverage achieved in various orders. The graph clearly infers a higher code coverage for fuzzing using random fuzz driver.

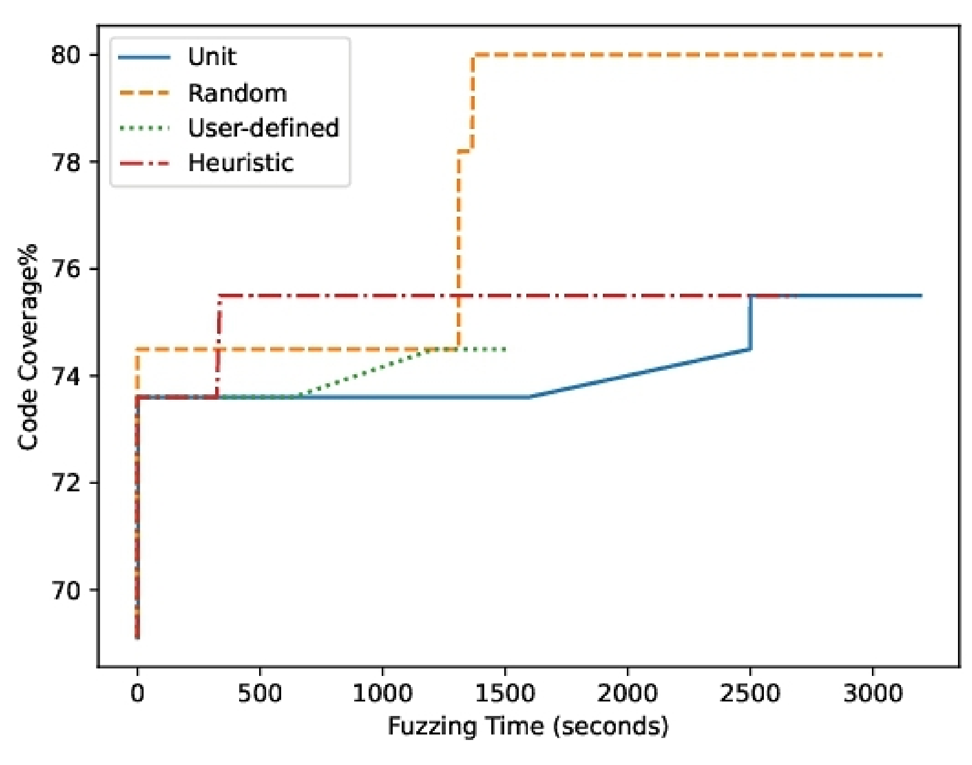


Figure 8: Coverage with various drivers

As discussed earlier, we ran fuzz tests outlining four cases. They are unit-level, heuristics-based, random sequencing of APIs and user-defined sequence based on graphs. Following are the observations which answer the research question **RQ2** related to ***effectiveness*** of different drivers.

- The code coverage was higher in the case of random sequencing of APIs than with the other strategies since it considers the stateful-ness of variables and evaluates many varied sequences of APIs if not all.
- The buffer-overwrite bug (CWE-787) could not be triggered except in handful of fuzz runs where the time to incident was very close to 1 hour because it had to satisfy the rigorous condition (data1 > 15 and data1 <= 20) where *data1* is an input to the API that is derived from the semi-random input generated by the fuzzer. Hence, the probability of *data1* having the value within the mentioned range is very low which makes it hard for the fuzzer to identify this bug. With a more lenient condition, the bug was easily triggered. The bug with the same condition is expected to be set off if the fuzz duration is extended. The other bugs were triggered within the test duration while testing with all the strategies.
- The sequence bugs are not triggered by unit-level and heuristic based fuzz tests as expected. Because these approaches do not support persisting the state variables across API calls. The random sequencing of API invocation triggered both the sequence bugs. The user-defined approach triggered one of the sequence bugs as expected, since only one of the bugs scattered along the valid sequence of API calls.
- Unit-level and heuristic-based fuzz tests took less time to identify the bugs that are not scattered across APIs and uncorrelated to stateful variables. The heuristics based approach here is designed only with respect to ranking of APIs on the basis of lines of code and could be modified to consider other parameters such as number of easily reachable code blocks in an API for better code coverage.
- Both random sequencing and user-defined drivers should be preferred for sequence-based testing as they consider the stateful variables. Though in the former case it might take more time for the bugs to be triggered, in the latter case the developer should possess sufficient knowledge about the smart contract code in order to supply correct API sequences as a graph for fuzzing.
- From the observations, we can infer that even though random sequence based fuzz driver is slower, it can cover maximum code and trigger maximum bugs in the provided time for the experiment. Thus random sequence fuzzing is useful and it should not be ruled out from fuzz testing.

5.5 Effectiveness Evaluation Based on Mann-Whitney U Test

We conducted Mann-Whitney U test to answer the research question **RQ3**, in order to statistically differentiate on the effectiveness and significance between the results of different fuzz drivers we used, that is, unit-level, random, user-defined and heuristics based. Since we carried out five runs for each fuzz driver experiment, we combined the incident times of bugs from each such runs and conducted the Mann-Whitney U test. A *p-value* of 0.00785 implies a significant difference between the results of random fuzz driver and unit-level driver. *p-values* from other tests did not infer significant differences. Similarly, considering the p-values in coverage calculation tests, we again see significant difference in random fuzz driver test versus all other fuzz drivers.

5.6 Other Observations and Discussions

In this section, we describe a few generic observations and discussions.

5.6.1 Seeds for Fuzzing Evaluation. Unlike other fuzzers, go-fuzz[28] doesn't necessarily require any seed to initiate the fuzzing process and uses an empty seed by default. We understand that providing a good seed to start with can improve the efficiency of fuzzing significantly, however, we haven't included identifying good seed in our scope; we limited ourselves to research on the interventions based on state variables and API sequence-based fuzzing.

5.6.2 Mapping of Crashing Inputs to Unique Bugs. Go-fuzz [28] produces stack traces / hashes on crashing inputs which provides information on the type of bug and lines of code responsible for crash. We also triaged the crashing inputs real-time on the source code and did not find any discrepancy between the stack trace reported and the actual *"ground truth"* [20]. However all the crashing inputs did not result in unique bugs as reported by the fuzzer; they were deemed interesting preferably due to coverage profile. We have reported the unique bugs after triaging in Table2.

5.6.3 Manual vs Automatic Fuzz Driver Generation. Manually handling the sheer volume of smart contracts and each of their APIs in enterprise settings for testing using their corresponding drivers is laborious and error prone. The steps for automatic generation of fuzz drivers have been accumulated from manually written fuzz driver experiments, handling of developer induced mistakes, and studying of different smart contract API patterns.

5.6.4 Availability of Benchmarks. All the sample Golang HLF smart contracts used for our experiments are from open source community and they are not complex when compared to real-world smart contracts (which implement supplychain workflow or payment processing etc.). Hence, we used a simple graph without any cycles to represent valid API sequences, as input to the user-defined sequence testing. The availability of complex smart contracts and thus representing them as multi-graphs is a viable possibility and an exciting future direction of work.

5.6.5 Importance of Negative Testing in Invalid Sequence Based Fuzzing. In case of user-defined sequence testing, we have also considered performing negative testing (that is, testing of invalid sequences) using a complement of the user-defined graph. However, the resultant multi-graph is complex and has cycles or loops and we scoped this work only to simple graphs. The random-sequence driver that was generated by *SmartFuzzDriverGen* produces many of the invalid sequences due to which *Sequence Bug 2* was triggered as shown in Table2. The consideration of the whole state space of invalid sequences and thereby fuzz testing them all could be inefficient and require significant amount of time. Improving the efficiency by ruling out certain combinations can be a future direction of work.

5.7 Threats to Validity

In this section, we will discuss the internal and external threats to validity of our experiments.

5.7.1 Internal Threats to Validity. The motivating example that we used is a simple one with only six APIs and the prototype implementation was not a complete functionality implementation; it was a implementation meant for implanting only bugs. Due to the small number of lines of code in each API, the heuristics driver based on number of lines ended up assigning multiple APIs the same rank. Hence, the order in which the individual APIs getting called changed between multiple runs. This could affect the results of the heuristics based driver.

5.7.2 External Threats to Validity. We considered loop-free automata for our motivating example workflow. However, in real world scenarios, the smart contracts can be more complex and multiple valid and invalid sequences can exist. This can increase the complexity of the smart contract significantly and the fuzzing can become slow to run sequence based drivers. One way to mitigate this problem is to configure the sequence length to a lower value. This can lead to missing of some APIs in a sequence. Striking the right balance between sequence length and coverage speed could be a challenge.

6 RELATED WORK

Almakhour et al. in [1] classified the verification tools/techniques into formal verification for correctness and vulnerability detection for security assurance. [6][30] discusses on the vulnerabilities such as risks due to non-determinism, range-query risks affecting Hyperledger Fabric smart contracts. Beckert et al. used KeY a semi-interactive theorem prover for verification of Hyperledger fabric smart contracts written in Java [3]. There are static analysis tools like Chaincode Scanner [19] and revive-cc [27] for smart contracts, however the former is a proprietary and latter is outdated. In an enterprise setting of verifying smart contracts before scaling applications into production, the fuzzing method is mostly preferred [24] due to its advantages of low resource requirements, easy scalability and faster triggering of surface bugs. Exploring several surface bugs in verifying the smart contracts through fuzzing has been the motivation for this work.

There are automated fuzz driver generation tools for c/c++ languages namely FUDGE [2], FuzzGen [18], and IntelliGen [32]. FUDGE uses client code that is based on the library (to be fuzzed) to learn valid usage of the library and uses them to generate drivers. It requires manual intervention to select effective fuzz drivers. FuzzGen on the other hand needs testcases to generate fuzz drivers. IntelliGen considers all APIs and generates fuzz drivers for them.

Coming to HLF smart contracts, we come across manual efforts like HFContractFuzzer [7] where the authors have changed the mutation scheme of the fuzzer to selectively fuzz for bugs for hyperledger fabric smart contracts, afuzzer [4] provides signals and alerts for chaincode quality control. These efforts simply demonstrate how the fuzz driver can be written using test environment, that too for older versions/types of smart contracts. However, there exists no tools for generating fuzz drivers for code written in Golang.

SMARTIAN paper [5] proposes a way to automatically identify valid API sequences using the data flow of state variables. However, this method will not work for Hyperledger Fabric smart contracts as, in Ethereum smart contracts, the variables defined in global scope are state variables. However, in HLF smart contracts, any variable can be put into the world state as a key value pair using the platform API *PutState()*. Hence, it is hard to identify which local variable is holding the key for which state variable.

Jens-rene et al. in [13] discuss smart contract compiler called hardening contract compiler (HCC) to harden Ethereum smart contracts by introducing additional checks. HCC models control-flows and data-flows of a given smart contract by using a code property graph (CPG). The authors of HCC evaluated it with Hyperledger Fabric also by translating the vulnerable functions from Solidity smart contracts to Golang and then applying HCC on the resulting smart contract. However, unlike fuzzing, this approach is meant to identify known vulnerabilities integer overflow/underflow and reentrancy bugs.

7 CONCLUSIONS AND FUTURE WORK

We presented *SmartFuzzDriverGen*, a framework to automatically generate fuzz drivers for Hyperledger Fabric smart contracts written in Golang.

We validated the framework by conducting experiments with a set of smart contracts. In order to enhance the effectiveness of fuzzer for coverage and time to trigger bugs (including bugs dependent on state variables), we introduced our strategies of generating unit-level, heuristic-based, and sequence-based (random and user-defined) fuzz drivers. When compared to Ethereum, the number of HLF smart contracts available in the public repositories are less. Hence, we evaluated our framework and strategies using a handful of smart contracts from HLF Samples repository and by synthesizing a buggy smart contract as discussed in [26]. From the observations, we infer that unit level fuzz drivers as well as heuristics based ones are sufficient to trigger bugs which are not scattered across multiple APIs. To trigger bugs which are dependent on multiple APIs being called in a particular sequence or out of sequence, sequence-based fuzz drivers are more suitable. As depicted by our observations on effectiveness and statistical evaluation, random-sequence based fuzz driver though slower, can cover maximum code and trigger maximum bugs when sufficient time is provided. It also covers the positive as well as (partly) negative testing considering the timing and resource constraints. If the developer can provide a set of valid API sequences in the form of a graph, we can use the user-defined fuzz driver as it can unearth sequence bugs as well as identify bugs relatively faster than random-sequence method.

We are planning to measure the effectiveness of the generated drivers on real-world smart contracts, in terms of code coverage and bug finding abilities. Also, this work can be extended to support smart contract written in other programming languages. The fuzz driver generation strategies, unit-level, heuristics-based, random-sequence based and user-defined sequence based could be used in a hybrid form for more efficiency. Improving the efficiency of negative testing by ruling out certain combinations, synthesis and planting of bugs to validate the performance of drivers can be some of the future works. Golang supports fuzzing natively since v1.18 [14]. Testing the strategies with native fuzzing can open possibilities like ease in writing drivers, better performance. Other testing strategies such as static analysis and formal verification approaches can also be employed in conjunction with fuzzing. Automatic smart contract repair can be another potential future direction.

REFERENCES

[1] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2020. Verification of smart contracts: A survey. *Pervasive and Mobile Computing* 67 (2020), 101227. Navigate to

[2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985. Navigate to

[3] Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. 2018. Formal specification and verification of Hyperledger Fabric chaincode. In *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM*. 44–48. Navigate to

[4] Jialiang Chang. 2020. *Software Quality Control Through Formal Method*. Western Michigan University. Navigate to

[5] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. Navigate to

[6] Ahaan Dabholkar and Vishal Saraswat. 2019. Ripping the fabric: Attacks and mitigations on hyperledger fabric. In *International Conference on Applications and Techniques in Information Security*. Springer, 300–311. Navigate to

[7] Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. Hfcontractfuzzer: Fuzzing hyperledger fabric smart contracts for vulnerability detection. In *Evaluation and Assessment in Software Engineering*. 321–328. Navigate to

[8] Hyperledger Fabric. 2022. Hyperledger Fabric Samples. **https://github.com/hyperledger/fabric-samples (https://github.com/hyperledger/fabric-samples)**. Navigate to

[9] Hyperledger Fabric. 2022. World-State. **https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html#world-state (https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html#world-state)**. Navigate to

[10] Josselin Feist. 2018. Contract upgrade anti-patterns. **https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/ (https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/)**. Navigate to

[11] Klint Finley. 2016. A $50 Million Hack Just Showed That the DAO Was All Too Human. **https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human (https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human)**. Navigate to

[12] ForAllSecure. 2022. The Roles of SAST and DAST and Fuzzing in Application Security. **https://forallsecure.com/blog/sast-and-dast-in-application-security (https://forallsecure.com/blog/sast-and-dast-in-application-security)**. Navigate to

[13] Jens-Rene Giesen, Sebastien Andreina, Michael Rodler, Ghassan O. Karame, and Lucas Davi. 2022. Practical Mitigation of Smart Contract Bugs. In *Arxiv*. 1–17. Navigate to

[14] Golang. 2022. Go Native Fuzzing. **https://go.dev/security/fuzz/ (https://go.dev/security/fuzz/)**. Navigate to

[15] Dan Goodin. 2021. Really stupid “smart contract” bug let hackers steal $31 million in digital coin. **https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance (https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance)**. Navigate to

[16] Google. 2022. gofuzz. **https://github.com/google/gofuzz (https://github.com/google/gofuzz)**. Navigate to

[17] LLVM Compiler Infrastructure. 2022. libFuzzer. **https://llvm.org/docs/LibFuzzer.html (https://llvm.org/docs/LibFuzzer.html)**. Navigate to

[18] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287. Navigate to

[19] Tobias Kaiser. 2018. Chaincode Scanner. **https://hgf18.sched.com/event/G8rZ/security-vulnerabilities-in-chaincode-tobias-kaiser-chainsecurity (https://hgf18.sched.com/event/G8rZ/security-vulnerabilities-in-chaincode-tobias-kaiser-chainsecurity)**. Navigate to

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138. Navigate to

[21] OpenZeppelin. 2022. Upgrading smart contracts. **https://docs.openzeppelin.com/learn/upgrading-smart-contracts (https://docs.openzeppelin.com/learn/upgrading-smart-contracts)**. Navigate to

[22] Alfrick Opidi. 2022. How To Address SAST False Positives In Application Security Testing. **https://www.mend.io/resources/blog/resources-blog-sast-false-positives/ (https://www.mend.io/resources/blog/resources-blog-sast-false-positives/)**. Navigate to

[23] Santiago Palladino. 2017. The Parity Wallet Hack Explained. **https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7 (https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7)**. Navigate to

[24] Siddhasagar Pani, Harshita Vani Nallagonda, Saumya Prakash, R Vigneswaran, Raveendra Kumar Medicherla, and MA Rajan. 2022. Smart Contract Fuzzing for Enterprises: The Language Agnostic Way. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 1–6. Navigate to

[25] Sergey Petrov. 2017. Another Parity Wallet hack explained. **https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c (https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c)**. Navigate to

[26] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 224–234. Navigate to

[27] sivachokkapu. 2020. Revive - CC. **https://github.com/sivachokkapu/revive-cc (https://github.com/sivachokkapu/revive-cc)**. Navigate to

[28] Dmitry Vyukov. 2022. go-fuzz. **https://github.com/dvyukov/go-fuzz (https://github.com/dvyukov/go-fuzz)**. Navigate to

[29] Wikipedia. 2022. Fisher-Yates shuffle algorithm. **https://en.wikipedia.org/wiki/Fisher-Yates_shuffle (https://en.wikipedia.org/wiki/Fisher-Yates_shuffle)**. Navigate to

[30] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. 2019. Potential risks of hyperledger fabric smart contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 1–10. Navigate to

[31] Michal Zalewski. 2022. American Fuzzy Lop (AFL). **https://github.com/google/AFL (https://github.com/google/AFL)**. Navigate to

[32] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327. Navigate to

ISEC 2023, February 23–25, 2023, Allahabad, India

© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0064-4/23/02…$15.00.
DOI: **https://doi.org/10.1145/3578527.3578538 (https://doi.org/10.1145/3578527.3578538)**