Computer Systems Science & Engineering
DOI: 10.32604/csse.2023.027680
Article

Tech Science Press

Formal Verification Platform as a Service: WebAssembly Vulnerability Detection Application

LiangJun Deng[1], Hang Lei[1], Zheng Yang[1], WeiZhong Qian[1,*], XiaoYu Li[1], Hao Wu[2], Sihao Deng[3], RuChao Sha[1] and WeiDong Deng[4]

[1]School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 610054, China
[2]School of Physics, University of Electronic Science and Technology of China, Chengdu, 611731, China
[3]ICB-PMDM-LERMPS UMR 6303, CNRS, UTBM, Université de Bourgogne Franche-Comté, Belfort, 90010, France
[4]Chengdu Railway Science and Technology Innovation Co. LTD, Chengdu, 640041, China
*Corresponding Author: WeiZhong Qian. Email: wzqian@uestc.edu.cn
Received: 24 January 2022; Accepted: 13 May 2022

Abstract: In order to realize a general-purpose automatic formal verification platform based on WebAssembly technology as a web service (FVPS), which aims to provide an automated report of vulnerability detections, this work builds a Hyperledger Fabric blockchain runtime model. It proposes an optimized methodology of the functional equivalent translation from source program languages to formal languages. This methodology utilizes an external application programming interface (API) table to replace the source codes in compilation, thereby pruning the part of housekeeping codes to ease code inflation. Code inflation is a significant metric in formal language translation. Namely, minor code inflation enhances verification scale and performance efficiency. It determines the efficiency of formal verification, involving launching, running, and memory usage. For instance, path explosion increases exponentially, resulting in out-of-memory. The experimental results conclude that program languages like golang severely impact code inflation. FVPS reduces the wasm code size by over 90%, achieving two orders of optimization magnitude, from 2000 kilobyte (KB) to 90 KB. That means we can cope with golang applications up to 20 times larger than the original in scale. This work eliminates the gap between Hyperledger Fabric smart contracts and WebAssembly. Our approach is pragmatic, adaptable, extendable, and flexible. Nowadays, FVPS is successfully applied in a Railway-Port-Aviation blockchain transportation system.

Keywords: WebAssembly; formal verification; blockchain; smart contract

1 Introduction

Background: WebAssembly [1–3] is an efficient and lightweight instruction set, which perfectly supports all types of central processing units (CPU), and it has been widely concerned by blockchain technology [4]. The eWASM team has set out to integrate WebAssembly on Ethereum in the next generation. Both Dfinity and enterprise operation system (EOS) [5] have chosen WebAssembly to enhance their execution performance. From 2011 to now, because of potential security vulnerabilities, more than 200 events have happened in blockchain, resulting in over 4 billion dollars in losses. The whole blockchain industry has been aware of vulnerability detection other than traditional unit tests before smart contracts [6] are deployed on a blockchain platform [7–10]. WebAssembly smart contract has also become a new urgent critical security problem to be solved. **Related Work:** The traditional software testing methodology [11] cannot guarantee smart contracts' correctness and high reliability. Formal verification [12] has been proposed in the blockchain field involving Solidity [13–15], ethereum virtual machine (EVM) bytecode [16–20], and EOS [21–22]. On one side, different verification frameworks commonly only support one particular language of smart contracts. It is not suitable for the blockchain with multiple smart contract languages, for instance, Hyperledger Fabric blockchain. On the other side, WebAssembly as an official standard of the World Wide Web Consortium (W3C) is the next generation instruction of smart contracts, and it is going to be massively applied to the blockchain smart contract field. Finally, Hyperledger Fabric is the famous mainstream consortium blockchain with a complex high-level language in smart contracts such as golang and C/C++. Hence, for the time being, there is no relevant research using WebAssembly instruction as intermediate representation code to verify Hyperledger Fabric [23–25] smart contracts. This work is fascinating for the sake of the factors mentioned above.

Contributions: This work can detect vulnerabilities of general-purpose programs written by the programming language of c, c++, rust, golang, etc. Furthermore, it can verify the smart contract of Hyperledger Fabric in real-time online. The major contributions of this work consist of three parts: (1) **FVPS framework:** provides an automatic online verification service, it implements a complete process of compilation, optimization, symbolic execution, verification work, and builds a virtual environment model of the Hyperledger Fabric in memory to simulate the distributed mechanism of blockchain for vulnerability detections of smart contracts. (2) **An optimized translation methodology for formal verification:** In reality, we discover that target program files are usually embedded by the targeted compilation platform with redundant codes such as golang runtime, which is useful based on the specific platform but unnecessary for the formal verification. Because the purpose of the formal language differentiates the standard program language, almost all program languages have the problem of a redundant compilation. In order to solve the problem, this work offers an optimized compilation translation model to prune redundant codes. We utilize an external API table as symbols to replace the part of housekeeping codes to ease code inflation. Then, it discards meaningless expression nodes to achieve the optimizations facilitating the efficiency of the whole subsequent work. (3) **A memory-based virtual Hyperledger Fabric blockchain runtime environment model:** The essence of blockchain is a decentralized distributed technology. It is hardly impossible to verify smart contracts in the whole distributed system environment. Mainly, in addition to distributed features, the Hyperledger Fabric blockchain for itself adopts an excellent logic separation architecture, which led to smart contracts having to access remote data storage nodes via its google remote procedure call (gRPC). However, this type of architecture is just a fatal design for formal verification, which is more difficult to abstract model than other blockchain platforms. This work simulates a memory-based virtual blockchain runtime environment and abstracts a data access model in memory to replace blockchain storage nodes. Smart contracts directly invoke gRPC interfaces to read/write data in our local machine's memory instead of remote machines. Besides, this model is also applicable to other distributed systems for formal verification.

Section 2 discusses the problems of the existing compilation method, such as official golang-build and low level virtual machine (LLVM), and introduces the framework of FVPS. Section 2.1 elaborates the translation methodology for formal verification, especially, we discover a significant optimization about housekeeping codes, as yet it is very effective in golang program language, and also we apply this methodology to Hyperledger Fabric smart contracts. Section 2.2 explains a five-layer sandbox model to simulate the distributed Hyperledger Fabric blockchain system. In experimental section 3, some schemes are designed to justify the effect of our optimization methodology. Finally, through web hyper text transfer protocol (HTTP), we send some golang/c/c++/rust source program codes to our verification service. Similarly, in our practical project, some smart contracts of the Hyperledger Fabric respectively gain an automated detection report from FVPS.

2 FVPS FrameWork

FVPS is divided into three independent services. In the first part, the interface service receives the different language source program codes via the web as a new verification project and stores them in an src folder. In the second part, the optimization compilation service compiles project source files from the src folder into WebAssembly programs written into a wasm folder with the suffix wasm. In the process, the WASM Manager distinguishes different project wasm module relationships with a user-defined project name. In the third part, the verification service consists of a wasm virtual machine and a symbolic execution tool with Mircosoft Z3Prover. The wasm virtual machine reads all the wasm module files from WASM Manager, and the symbolic execution engine begins to verify the target programs to find out whether it is vulnerable. Indeed, we have to admit that our current vulnerability detection capabilities are not mature enough because the combination of WebAssembly and blockchain environment is complex, involving massive workloads. The priority goal of this work is to construct a whole integrated WebAssembly formal verification platform to eliminate the gap between blockchain smart contracts and WebAssembly. Hence the complicated detection capabilities will be considered in the subsequent work. Finally, the verification service outputs a responding report of detection results. The overall architecture is shown in Fig. 1.

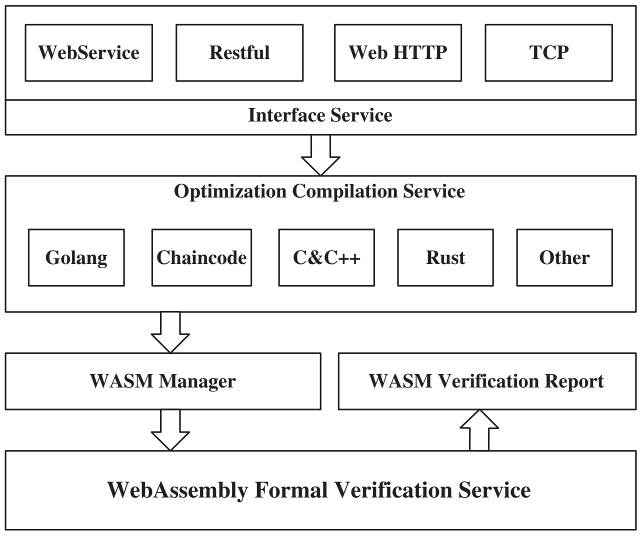


Figure 1: The architecture of this framework

In this work, when FVPS starts work, one user can send source program codes to the interface service via the APIs of FVPS software development kit (SDK). Next, FVPS predicts the programming language of source codes and automatically completes the whole verification work with unattended operations. Several seconds later, the user is able to gain a detection report from FVPS. There is a clear flow graph of this work illustrated in Fig. 2.

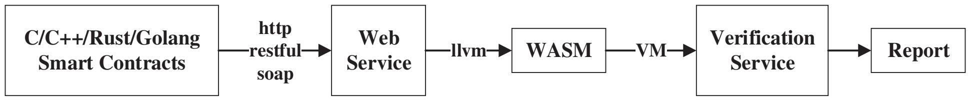


Figure 2: The verification process of wasm vulnerability reports

Verification Service is an essential service in the entire process of this platform involving two vital factors. Firstly, to efficiently detect vulnerabilities by symbolic execution technology, the size of object files is required to be as small as possible. Symbolic execution will simulate the path exploration in logical branches resulting in memory and time consuming, even depleting the memory along with the increasing paths. In reality, Google has offered the go-llvm project to compile golang source codes into the intermediate representation of LLVM. According to the official website, the go-llvm is just used as a benchmark of efficiency comparison, it does not support the compilation of WebAssembly, but the go-llvm provides a new idea to the officials of LLVM and golang. Hence golang officially supported the compilation of the WebAssembly platform next, and LLVM began to support WebAssembly from version 8.0 in experimental mode. Unfortunately, the existing methods all have the problem of inflating the target program file. No matter how many golang source codes there are, the final binary/assembly file size occupies at least 2 megabytes (MB). In the format of intermediate representation, even soars to 73 MB. Secondly, the symbolic execution technology suggests simulating a virtual runtime environment corresponding to the target platform. As is well-known, WebAssembly pertains to Web3.0 technology, which is initially designed for a web browser in a javascript environment. Therefore, the WebAssembly of golang and LLVM only runs in a web browser with the javascript technology. If this work uses the existing official technology, this situation is not suitable for formal verification. WebAssembly Formal verification presents many challenges, so we propose a novel formal verification translation methodology to solve all the aforementioned problems.

2.1 Formal Verification Translation Methodology

This methodology designs a multi-pass compilation architecture consisting of a preprocessor, lexer, parser, semantic tree, abstract syntax tree (AST), and intermediate representation (IR). In the phase of the preprocessor, this work builds a function-based invocation chain relation table corresponding with the including relation of housekeeping files such as "include<stdio.h>" or "import fmt" etc. Next, an extendable API table based on regular expression is added to our translation. The lexer shall match tokens with the external API table to compile the tokens as symbols instead of housekeeping codes. To clearly explain the optimized relationship between the API table and the housekeeping codes, we define some basic symbols of this work and an optimized translation process in the subsequent section.

Firstly, this work designs some symbols in Tab. 1 to describe the source program code using formal language.

S denotes the total source codes in our definitions, consisting of user-written codes and compiler-appended codes called "housekeeping codes." For all we know, the housekeeping code is a necessary padding part for any compiler to work correctly. Supposed $L1$-n denoted the expression of user-written clauses. Similarly, the Ln-m denoted the expression of compiler-appended clauses. This work describes it as follows.

$$S = \bigcup_{i=1}^{n+m} L_i = \underbrace{L_1 \cup L_2 \cup L_3 \ldots \cup L_n}_{user-written} \cup \underbrace{L_{n+1} \cup L_{n+2} \ldots \cup L_{n+m}}_{compiler-appended} \tag{1}$$

Table 1: Basic symbols of source program codes

S	Source codes.
L_i	logic expression clauses of S, consists of L^s and L^v.
L^s	$L^s \in$ Expressions of housekeeping imported by include or import.
L^v	$L^v \in$ Logic Expressions almost written by developers.
$f_j^s(s)$	Equivalent function format of L^s
$f_j^v(v)$	Equivalent function format of L^v
$f'_\alpha(s)$	Root functions of the recursive invocation chain, top-level function in source program codes, consists of several $f_j^s(s)$.
$Api_j^*(\varepsilon)$	APIs of formal verification runtime, functionality equals $f'_\alpha(s)$, implemented by formal verification virtual machine.

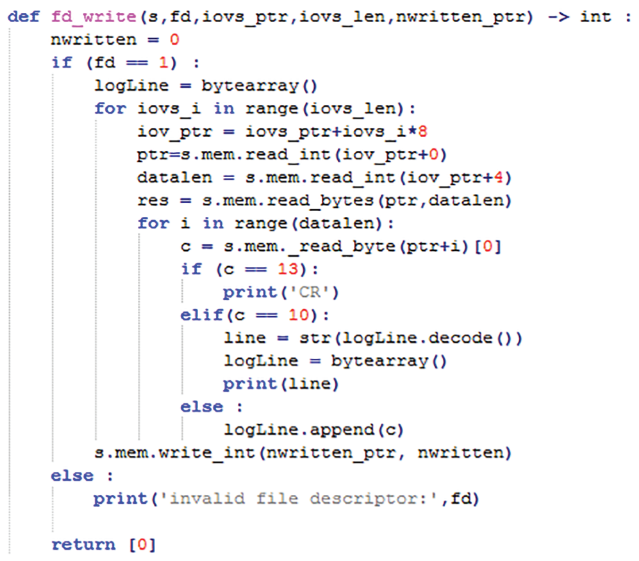
Secondly, to clarify our optimization method, this work splits S into user-written and compiler-appended expressions. It is easy to understand the user intention logic part without extra optimization space. On the contrary, the logic part should be kept as much as possible in the formal translation. In general, the amount number of compiler-appended housekeeping codes is ignored. Indeed, there is little performance affection in some program languages such as C/C++. Actually, the compilation result of different optimizers reveals that the target program is more or less affected by the housekeeping code. However, some program languages have a complex runtime environment, such as the golang language, which includes many APIs to uphold its program, even if the target program needs not to access them. In other words, some of the compiler's housekeeping codes are relatively redundant for one program. Generally speaking, it is automatically added by "include" or "import" grammar (such as a stdio header file, a golang fmt package.) in the preprocessing phase. For conveniently describing housekeeping code expressions, the source code Eq. (1) is concisely expressed by Eq. (2).

$$S = \bigcup_{i=1}^{n} L_i^v \bigcup_{j=1}^{m} L_j^s, (\underbrace{v \in Expression_{\log ic}}_{user-written}, \underbrace{s \in Expression_{\text{housekeeping}}}_{compiler-appended}) \tag{2}$$

Usually, most of the studies consider the performance affection of the housekeeping code to be slight, so the existed formal language translation methodology pays no attention to the housekeeping code. We supposed such a scenario: the user-written source codes are far less than the housekeeping codes. The housekeeping codes would become a great optimization question. First of all, we discuss whether the housekeeping code should be optimized. Along with the maturity of APIs, application binary interfaces (ABI) and runtime libraries, those parts are increasingly out of the verification question. Secondly, we analyze how to optimize the housekeeping code in formal verification translations. In Eq. (3), $Api_j^*(\varepsilon)$ denotes the APIs of FVPS, the functions $f_{\mathrm{j}}^s(x)$ and the expressions $\mathrm{L}_j^s$are equivalent. Taking golang as a typical example, the invocation chain of L_j^s is so deep for the sake of golang runtime. If those housekeeping functions can be bypassed, it can enhance the target program. This work proposed the target program across the golang runtime functions directly call functions from the virtual machine, namely, the housekeeping code can be removed. The following is an explanation of the missing function codes. The missing functions are replaced by symbols in an external API table without concrete codes in the compilation phase to compile the source code correctly. Meanwhile, our approach prunes the dead AST nodes, adopting the classical recursive descent parser (RDP) algorithm. The RDP algorithm is helpful in predictively looking up one function invocation relationship. Assume that a token pertains to the external API table. The compiler keeps only a symbol in the target program, pruning the related AST nodes. Indeed, the symbol is eventually upheld by our FVPS in runtime.

$$\mathrm{L}_j^s \Leftrightarrow f_{\mathrm{j}}^s(x) \Leftarrow Api_j^*(\varepsilon) \in FVPS. \tag{3}$$

To distinctly elaborate the APIs of logic expressions and housekeeping expressions, we take standard input/output (stdio) as an example. “stdio” is respectively denoted by “printf” in C language, “cout” in C++ language, “printf” in golang language, “println” in rust language. It will generate redundant AST nodes due to the implementation pattern of different platforms. In Fig. 3, the “fd_write” method is a function in FVPS to provide a symbol to the external API table. “fd_write” as a file descriptor interface can replace all analogous “print” symbols in housekeeping codes. In practice, this work gains a prominent effect in golang language for the sake of the excessive runtime codes.



```python
def fd_write(s,fd,iovs_ptr,iovs_len,nwritten_ptr) -> int :
    nwritten = 0
    if (fd == 1) :
        logLine = bytearray()
        for iovs_i in range(iovs_len):
            iov_ptr = iovs_ptr+iovs_i*8
            ptr=s.mem.read_int(iov_ptr+0)
            datalen = s.mem.read_int(iov_ptr+4)
            res = s.mem.read_bytes(ptr,datalen)
            for i in range(datalen):
                c = s.mem._read_byte(ptr+i)[0]
                if (c == 13):
                    print('CR')
                elif(c == 10):
                    line = str(logLine.decode())
                    logLine = bytearray()
                    print(line)
                else :
                    logLine.append(c)
        s.mem.write_int(nwritten_ptr, nwritten)
    else :
        print('invalid file descriptor:',fd)

    return [0]
```

Figure 3: File descriptor write interface implementation, written by python in formal verification platform to support the print function of different languages

Thirdly, L will be translated to an equivalent function. The source code can be described by the function ***F***(x).

$$F(x) = \bigcup_{i=1}^{n} f_i^v(v) \bigcup_{j=1}^{m} f_j^s(s). \tag{4}$$

According to Eq. (3), we can easily transform Eqs. (2) to (4). We design a top-level function called $f'_\alpha(s)$ to denote a sequence of related functions $f_j^s(s)$. For instance, “print” function as a library function is explicitly written by user, we define the “print” function as a top-level API. Under “print” function, there are still many functions in the runtime environment to uphold the “print” function. $f'_\alpha(s)$ is described as the following Eq. (5).

$$f'_\alpha(s) = \bigcup_{j=1}^{m} f_j^s(s), \alpha \in [1, 2, 3\ldots, n]. \tag{5}$$

Eq. (6) can be obtained from Eqs. (4) and (5). It implies $F_{(x)}$ consists of logic expression parts and some top-level functions f'_{α}.

$$F(x) = \bigcup_{i=1}^{n} f_i^{v}(v) \cup f'_{\alpha}(s), \alpha \in [1, 2, 3\ldots, n] \tag{6}$$

In the formal verification environment, $f'_{\alpha}(s)$ will be replaced by the functional equivalent APIs of FVPS. From Eqs. (3) and (6), we can get the final Eq. (7). It represents that the final program consists of the user-written logic functions and function symbols of FVPS.

$$F(x) = \bigcup_{i=1}^{n} f_i^{v}(v) \cup Api_j^{*}(\varepsilon), j \in [1, 2, 3\ldots, n]. \tag{7}$$

Namely, one symbol takes the place of a sequence of functions $f_j^{s}(s)$. It vividly reveals both the time- and space- complexity are significantly enhanced. In this case, this work can gain a derivation as shown in Eq. (8) that external APIs facilitate formal verification indeed in theory.

$$\begin{aligned}
&Api^{*}(\varepsilon) \Leftrightarrow f' \Leftrightarrow \left(f_1^{s} \cup f_2^{s} \cdots \cup f_{\mathrm{n}}^{s}\right)\\
&\Leftrightarrow \left(L_1^{s} \cup L_2^{s} \cdots \cup L_{\mathrm{n}}^{s}\right)\\
&\Rightarrow \mathrm{Expression}_{runtime} \Rightarrow HousekeepingCode \Leftrightarrow SourceCode_{include}\\
&\Rightarrow Optimiaztion\mathrm{Evaluation}(\mathrm{Api}) \nearrow\\
&\Leftrightarrow Complexity(SourceCode_{include}) \nearrow\\
&\Leftrightarrow Count(Node_{AST}) \nearrow
\end{aligned} \tag{8}$$

From Eq. (8), the derivation implies that the code inflation almost linearly increases along with the number of AST nodes. Our optimization effect is related to the original invocation depth. Namely, As $f'_{\alpha}(s)$ increase in complexity, our approach can replace more AST nodes with only a single symbol. In practice, we discover an empty golang main function causes 2000 KB in Linux. We conclude that the housekeeping codes named "go runtime" should be classified as L_{j}^{s}, which occupies almost equivalent to 2000 KB. Namely, the ultimate pruned effect can achieve close to 2 MB in Hyperledger Fabric smart contracts. Finally, we design a flexible, configurable API table based on regular expressions to determine whether prunes one subtree of AST. Similarly, our methodology can be applicable and compatible with other languages. In the experiment section, the practice result demonstrates the considerable effect of the golang programs and Hyperledger Fabric smart contracts. Our compilation result is less than 100 KB other than the official's 2000 KB.

2.2 Formal Verification Sandbox Model For Hyperledger Fabric

As a rule, a customized virtual machine can easily interpret the standard WebAssembly programs according to official standard WebAssembly instructions. Nevertheless, it suddenly becomes difficult in a blockchain distributed system. Hyperledger Fabric not only involves intricate interaction protocols among distributed nodes but also has a message mechanism in chaincode (smart contract) nodes so that after chaincode starts, nothing will be done unless it receives a chaincode message. Moreover, the Hyperledger Fabric smart contract consists of logic operation nodes and data storage nodes in multiple server nodes. The primary architecture of smart contracts is illustrated in Fig. 4.

Fig. 4 concisely shows the activated chaincode message relationship in the Hyperledger Fabric blockchain cyberspace. In formal verification, symbolic execution needs to simulate the architecture in its environment. This work summarizes a flow diagram in Fig. 5 to describe the smart contract mechanism of Hyperledger Fabric.

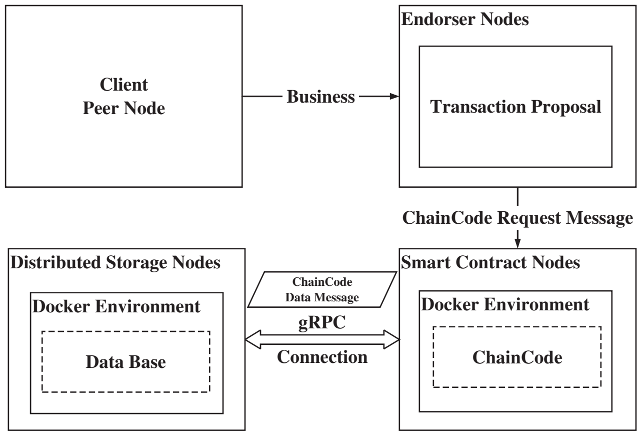


Figure 4: The logic separation architecture of the hyperledger fabric

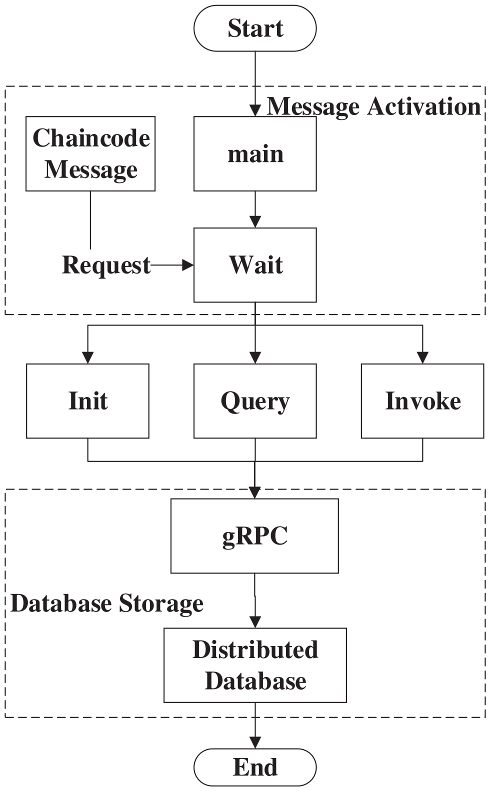


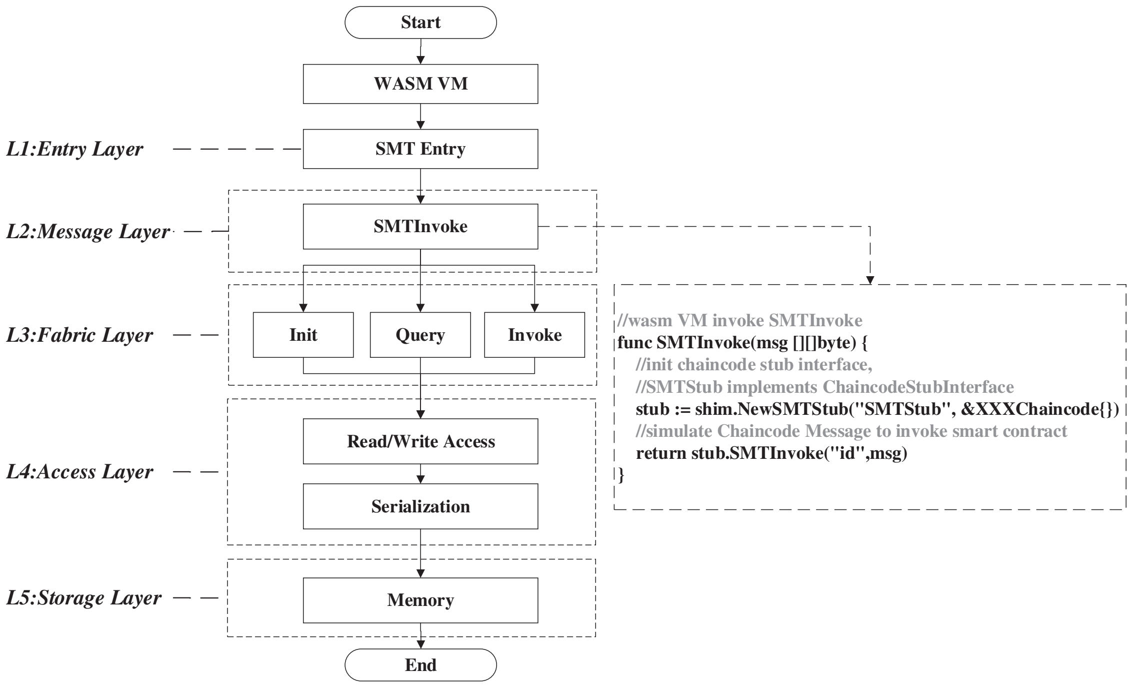
Figure 5: The flow diagram of Hyperledger Fabric smart contracts in chaincode node

In order to solve the problems of the distributed architecture, this work designs a five-layer model, as shown in Tab. 2.

According to the flow diagram in Fig. 5, this work reconstructs a hyperledger fabric smart contract sandbox model with five-layer hierarchies in Fig. 6. Message Layer implements a message-activated mechanism from a sandbox to replace the chaincode message mechanism. Entrypoint Layer is another new entrypoint of smart contracts. Formal verification starts from function SMTEntry instead of activating the original mechanism to avoid Hyperledger Fabric’s original main entrypoint. Therefore, it can be activated not only by a chaincode message but also by ***L***msg. Fabric Layer is the original blockchain layer. Certainly, it accesses a database by the stub's PutState interface and GetState interface. However, in this five-layer model, ***L***access is the actual implementation of PutState/GetState interfaces. ***L***access eventually read/write in the memory space of this smart contract sandbox other than a remote distributed database. Access Layer and Storage Layer eliminate the cross-node access and storage. The whole model fuses five layers into an enable formal verification functionally equivalent sandbox environment, corresponding to the distributed architecture in Fig. 5. Although in our formal verification model, the customized implementation is not the complete absolute replication of the distributed architecture, we shall firstly consider the smart contract detections are the primary target, not focus on whether the access layer is upheld by gRPC, whether the business data is stored in distributed databases or memory space. Secondly, modern software engineering technology is more robust than ever, and analogous transport/storage verification is not our purpose. Perhaps it is another research next.

Table 2: The definition of 5 layers

L_{msg}	Simulate chaincode message, activate smart contract to work by the SMTInvoke function of formal verification
L_{entry}	A new initial entrypoint for formal verification to start smart contracts
L_{fabric}	The Hyperledger Fabric function layer.
L_{access}	An access layer of input/output (I/O), RPC, read/write (R/W) APIs, etc. bridge original APIs to $L_{storage}$
$L_{storage}$	A memory-based storage model, accessed by L_{access}



```
//wasm VM invoke SMTInvoke
func SMTInvoke(msg [][]byte) {
    //init chaincode stub interface,
    //SMTStub implements ChaincodeStubInterface
    stub := shim.NewSMTStub("SMTStub", &XXXChaincode{})
    //simulate Chaincode Message to invoke smart contract
    return stub.SMTInvoke("id",msg)
}
```

Figure 6: The smart contract activation model of this work

Obviously, this paper has been done a great quantity of work in $\boldsymbol{L}_1$, $\boldsymbol{L}_2$, $\boldsymbol{L}_4$, and $\boldsymbol{L}_5$, involving a new ChaincodeStubInterface implementation called SMTStub, a fused complex flow mechanism, five-layer implementations, and an automated compilation task of Hyperledger Fabric smart contracts. Certainly, there are still enormous workloads in the WebAssembly-based virtual machine to sustain the sandbox model and eliminate the wasm javascript environment. Furthermore, constraint solver is a fundamental core component of formal verification, and this work chooses z3 prover from Microsoft Research.

3 Experiments & Analysis

In this section, this experiment prepared seven schemes in Tab. 3, compiled the programming source code involving C/C++/Rust(Scheme C(1–3)), golang(Scheme G(1–2)), Hyperledger Fabric smart contracts(Scheme SC(1–2)) to justify our optimized theory that housekeeping codes inflate the target program to a certain extent. Next, compared with official compilations of LLVM and golang, this experiment evaluates the optimized predictive effect of our methodology. At last, FVPS runs some examples to verify some simple Hyperledger Fabric smart contracts.

Table 3: Source code information of schemes

Scheme	Clause combinations	Format	Size(KB)
Scheme C_1	Printf	c/cpp	0.06
Scheme C_2	Printf +Function	c/cpp	0.14
Scheme C_3	Function	c/cpp	0.67
Scheme G_1	Printf + Function	go	0.12
Scheme G_2	Printf+Functions	go	0.59
Scheme SC_1	WriteOrder Contract Printf+Function	go	6.63
Scheme SC_2	WriteOrder Contract Printf+Functions	go	7.00

Firstly, this experiment writes some source codes involving one clause “printf hello world”, three methods: “add method”, “max method”, “collatz method”, and one smart contract of WriteOrder. This work will combine them into Scheme C, G, and SC in different languages.

As demonstrated in Fig. 7, the target program file size of Scheme C3 is less than 3 KB. It consists of only several logic methods without extra housekeeping files. On the contrary, the target file size of Scheme C1 is more than 10 KB caused of one “Printf” clause. Therefore, it is evident that the target program file size of Scheme C2 is the biggest one because of the combined source code, including one “Printf” clause and one logic method. As shown in Tab. 3, the source code file size of Scheme C3 is the biggest, but the target program size of Scheme C3 is the smallest among Scheme C(1–3). Because both Scheme C1 and Scheme C2 include the “stdio.h” housekeeping file, It is not difficult to infer that the “Printf” API causes code inflation. The entire program source codes shall be composed of housekeeping codes hidden in the compilation and visible source codes written by developers. These two factors commonly determine the final target program code complexity. As this paper proposed in the methodology section, although we obtain somewhat marginal fruition in the limited space, its application to Hyperledger Fabric smart contracts gains considerable achievements.

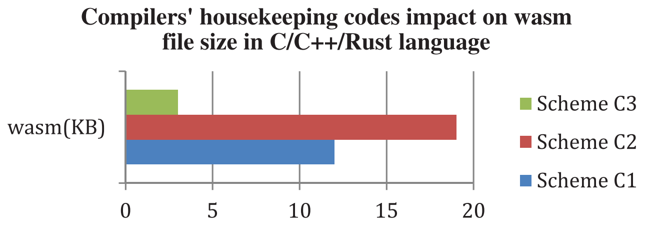


Figure 7: The target program code size of different source code combinations

Applying this formal verification translation methodology to golang and Hyperledger Fabric smart contracts are reasonable and pragmatic. As shown in Fig. 8, Scheme G(1–2) represents golang standard programs. The target wasm file size is approximate 2.2 MB, whether the compiler pertains to the golang official builder or LLVM. Similarly, Scheme SC(1–2) represents smart contracts in the golang language. The target program file size of hyperledger fabric smart contracts reaches close to 2.3 MB using the existing compilation tools. In this experiment, this work facilitates two orders of optimization magnitude. The final wasm file size reduces to less than 90 KB.

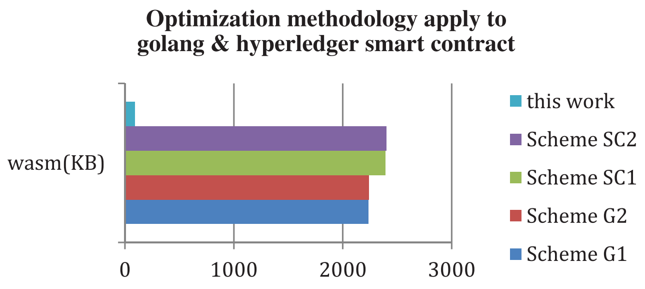


Figure 8: The wasm size of golang programs by default compilation of go build or LLVM

This work considers that the WebAssembly platform influences the final compilation result. This experiment respectively compiles Scheme G1, SC1 in the binary format on the Linux native platform through golang native build, as shown in Fig. 9.

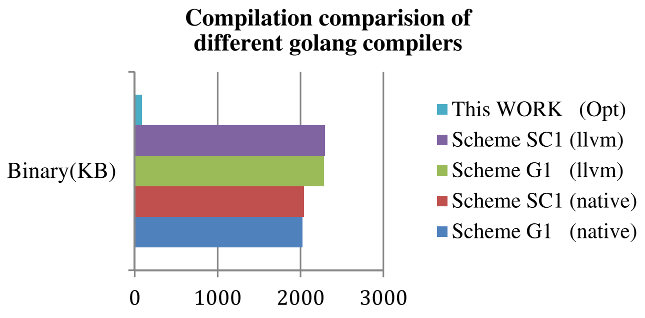
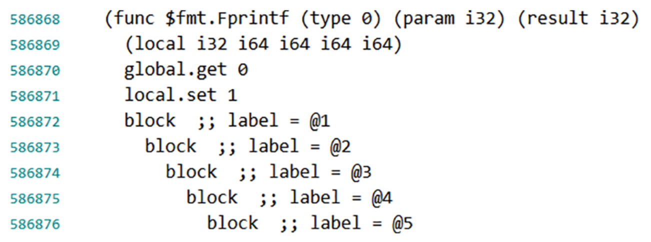


Figure 9: The file size of golang programs compiled by go native build compared with this work

Proof by facts, the golang original native binary compilation program occupies about 2 MB. The code inflation of the LLVM compilation exceeds the native platform, not up to 20%. Therefore, there are few impacts of different target compilation platforms in golang. It is a prominent contrast since this work optimizes the same source program codes to 90 KB. It also powerfully reveals that our formal verification translation methodology is efficient. This work analyses the readable text format of a wasm file in

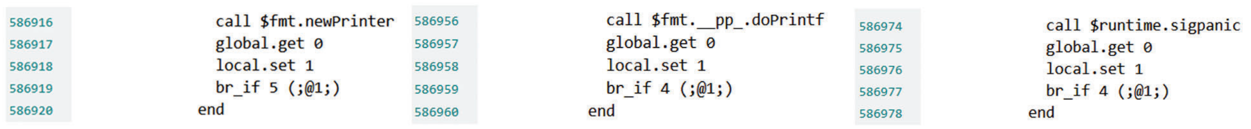
Fig. 10. For instance, one top-level function of the source program code is “$fmt.Fprintf,” which is the root function of the invocation function chains.



```
(func $fmt.Fprintf (type 0) (param i32) (result i32)
  (local i32 i64 i64 i64 i64)
  global.get 0
  local.set 1
  block  ;; label = @1
    block  ;; label = @2
      block  ;; label = @3
        block  ;; label = @4
          block  ;; label = @5
```

Figure 10: A root function of one invocation chain in a readable text format of WebAssembly

Along with one path in Fig. 10, the root function can respectively reach function “$fmt.newPrinter” in line number 586916, function “$runtime.sigpanic” in line number 586956, and function “$fmt.__pp_. doPrintf” in line number 586974, etc., given in Fig. 11.



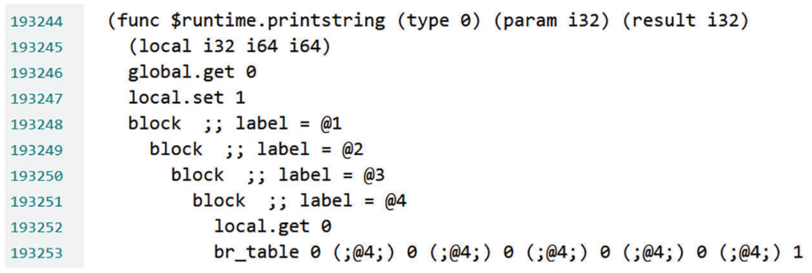
```
586916          call $fmt.newPrinter
586917          global.get 0
586918          local.set 1
586919          br_if 5 (;@1;)
586920        end
```

```
586956          call $fmt.__pp_.doPrintf
586957          global.get 0
586958          local.set 1
586959          br_if 4 (;@1;)
586960        end
```

```
586974          call $runtime.sigpanic
586975          global.get 0
586976          local.set 1
586977          br_if 4 (;@1;)
586978        end
```

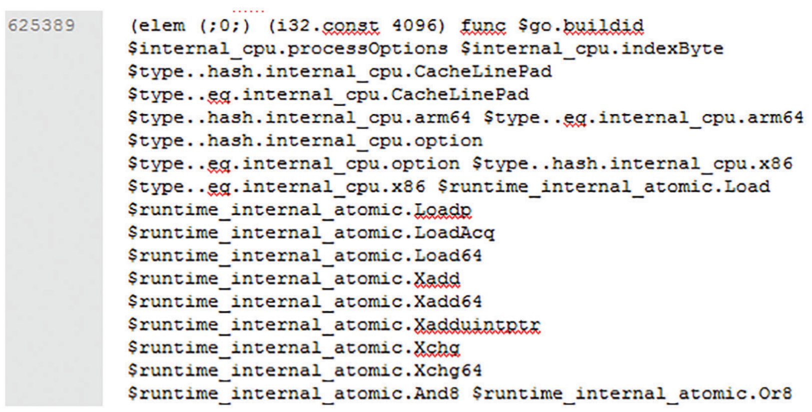
Figure 11: Three reachable intermediate functions in one invocation chain

Furthermore, “$runtime.printString” will be invoked by the upper functions in Fig. 12. Finally, all leaf functions in the invocation chain are provided by the “$runtime.Function*” functions, shown in Fig. 13.



```
(func $runtime.printstring (type 0) (param i32) (result i32)
  (local i32 i64 i64)
  global.get 0
  local.set 1
  block  ;; label = @1
    block  ;; label = @2
      block  ;; label = @3
        block  ;; label = @4
          local.get 0
          br_table 0 (;@4;) 0 (;@4;) 0 (;@4;) 0 (;@4;) 0 (;@4;) 1
```

Figure 12: One runtime function in one invocation chain



```
(elem (;0;) (i32.const 4096) func $go.buildid
$internal_cpu.processOptions $internal_cpu.indexByte
$type..hash.internal_cpu.CacheLinePad
$type..eq.internal_cpu.CacheLinePad
$type..hash.internal_cpu.arm64 $type..eq.internal_cpu.arm64
$type..hash.internal_cpu.option
$type..eq.internal_cpu.option $type..hash.internal_cpu.x86
$type..eq.internal_cpu.x86 $runtime_internal_atomic.Load
$runtime_internal_atomic.Loadp
$runtime_internal_atomic.LoadAcq
$runtime_internal_atomic.Load64
$runtime_internal_atomic.Xadd
$runtime_internal_atomic.Xadd64
$runtime_internal_atomic.Xadduintptr
$runtime_internal_atomic.Xchg
$runtime_internal_atomic.Xchg64
$runtime_internal_atomic.And8 $runtime_internal_atomic.Or8
```

Figure 13: Parts of golang runtime functions

This work can organize a simple invocation relation tree to expose the correlation between the root and invocation functions. It explicitly shows the function "$fmt.Fprintf" in Fig. 14.

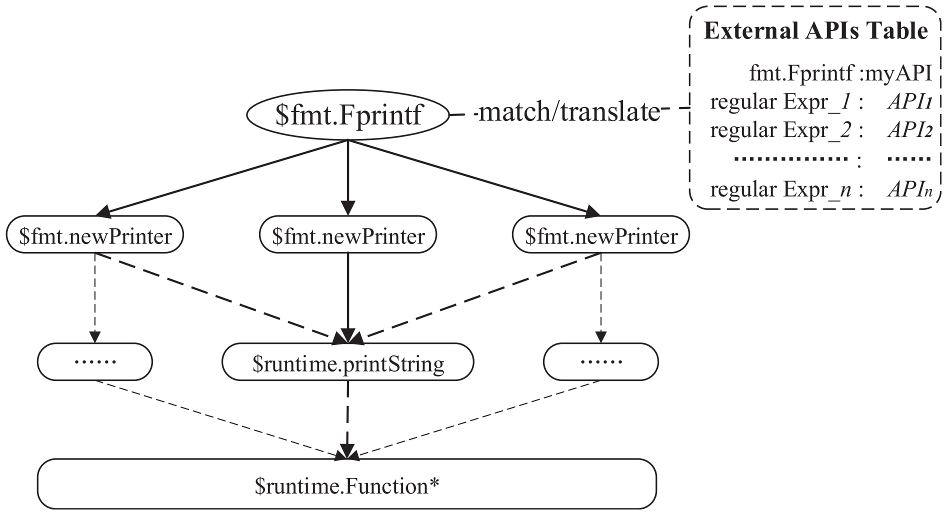
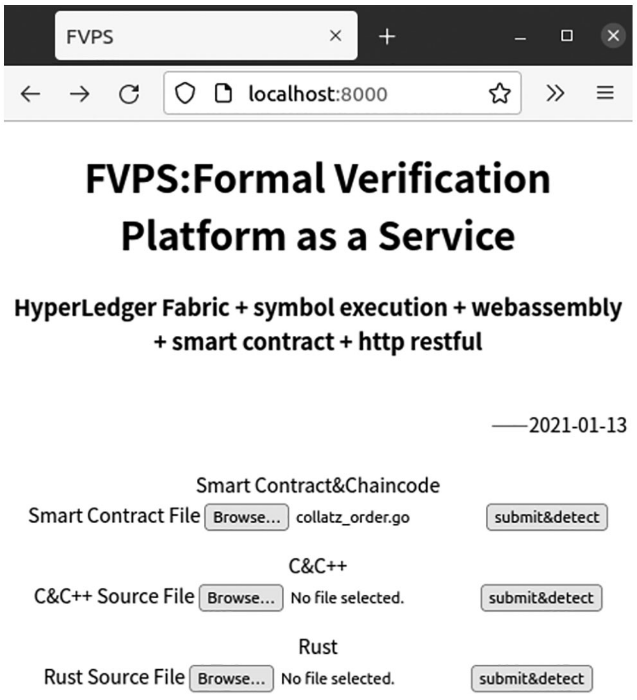


Figure 14: Depending relationship of one invocation chain

It is so easy to define a regular expression in our external API table to match "$fmt.Fprintf." When our formal verification translation methodology uses an external customized API to replace "$fmt.Fprintf," for one aspect, the source code analyzer can avoid analyzing more redundant housekeeping codes. For another, the optimizer can easily find out that the functions of "runtime" are unreachable dead codes, which are reasonable to be pruned by a compilation.



FVPS:Formal Verification Platform as a Service

HyperLedger Fabric + symbol execution + webassembly + smart contract + http restful

——2021-01-13

Smart Contract&Chaincode

Smart Contract File collatz_order.go

C&C++

C&C++ Source File No file selected.

Rust

Rust Source File No file selected.

Figure 15: A web front-end verification page of FVPS

Run Examples: In Fig. 15 the HTTP web server of this work is developed by Python FastAPI in the Conda environment. As so far, the webserver supports restful interfaces, web service description language (WSDL) interfaces based on simple object access protocol (SOAP), and a hyper text mark-up language (HTML) web page.

In Fig. 16, when users submit different source program code files to FVPS, they are stored in an src directory of the workspace. The target WebAssembly files are outputted into a wasm directory through our compilation.

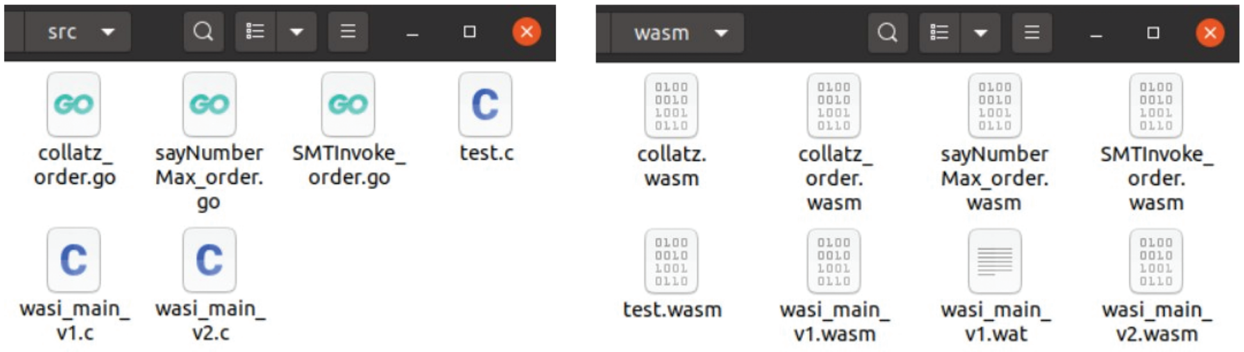
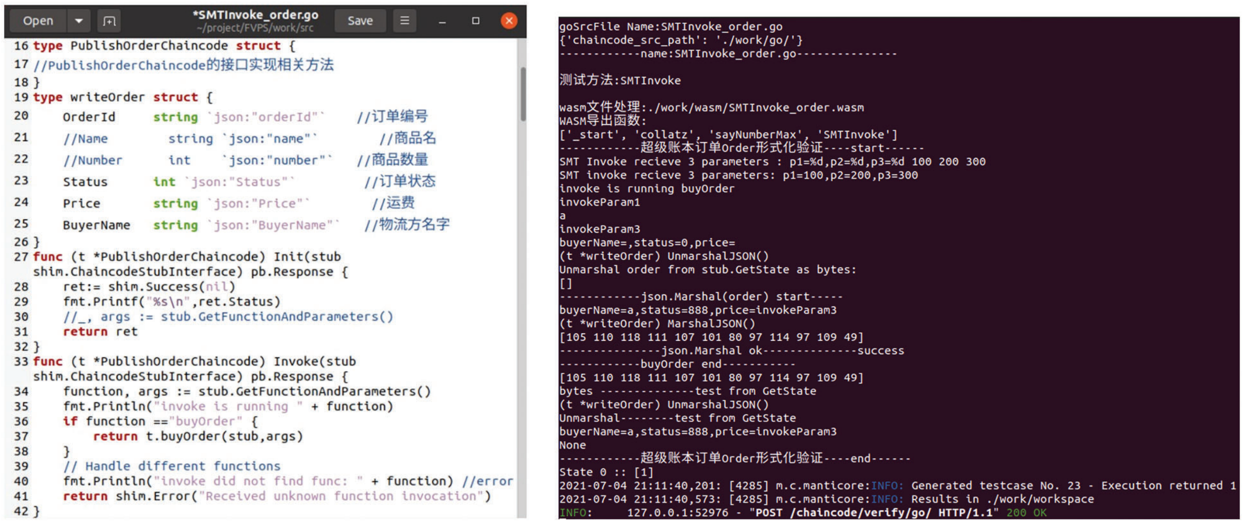


Figure 16: A source program code directory of FVPS and a wasm code directory of FVPS

Next, the formal verification service launches wasm files in a virtual machine byte by byte. Each operate code (op) instruction is interpreted and executed by python within a state. In formal verification, program execution is isolated in its memory space. Each group of inputted symbolic variables generates an instance. In a general way, symbolic execution is a non-deterministic polynomial (NP) problem. The traditional method suggests restricting inputted constraints or exploring in maximum limited duration. In this work, FVPS sets a five-minute limited time within a single thread to test our Hyperledger Fabric smart contract in Fig. 17.



```go
type PublishOrderChaincode struct {
//PublishOrderChaincode的接口实现相关方法
}
type writeOrder struct {
    OrderId     string `json:"orderId"`     //订单编号
    //Name        string `json:"name"`        //商品名
    //Number      int    `json:"number"`    //商品数量
    Status      int `json:"Status"`         //订单状态
    Price       string `json:"Price"`        //运费
    BuyerName   string `json:"BuyerName"`   //物流方名字
}
func (t *PublishOrderChaincode) Init(stub
shim.ChaincodeStubInterface) pb.Response {
    ret:= shim.Success(nil)
    fmt.Printf("%s\n",ret.Status)
    //_, args := stub.GetFunctionAndParameters()
    return ret
}
func (t *PublishOrderChaincode) Invoke(stub
shim.ChaincodeStubInterface) pb.Response {
    function, args := stub.GetFunctionAndParameters()
    fmt.Println("invoke is running " + function)
    if function =="buyOrder" {
        return t.buyOrder(stub,args)
    }
    // Handle different functions
    fmt.Println("invoke did not find func: " + function) //error
    return shim.Error("Received unknown function invocation")
}
```

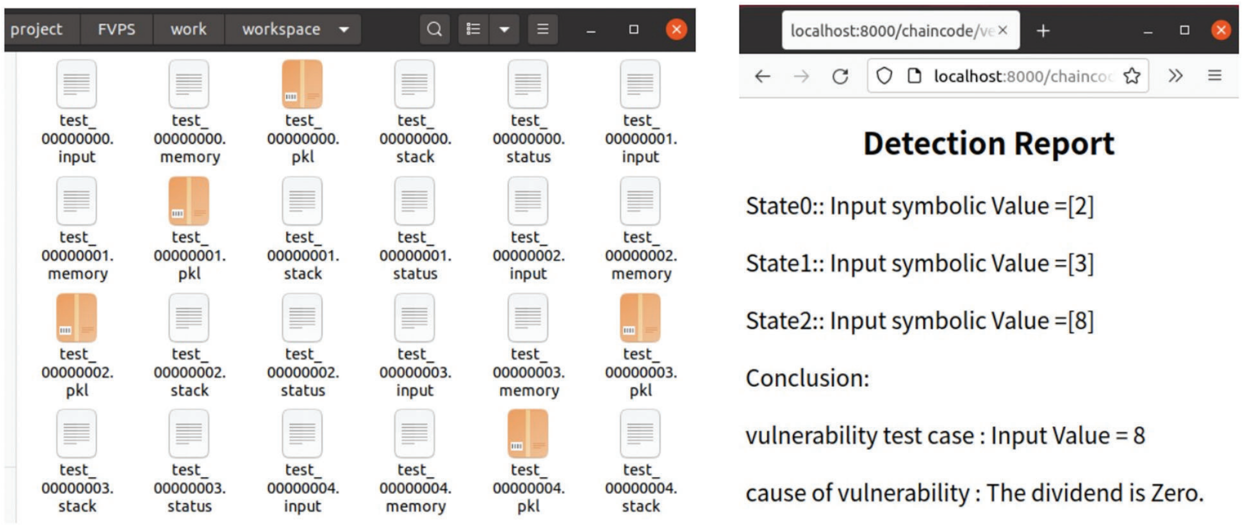
```
goSrcFile Name:SMTInvoke_order.go
{'chaincode_src_path': './work/go/'}
------------name:SMTInvoke_order.go---------------

测试方法:SMTInvoke

wasm文件处理:./work/wasm/SMTInvoke_order.wasm
WASM导出函数:
['_start', 'collatz', 'sayNumberMax', 'SMTInvoke']
------------超级账本订单Order形式化验证----start------
SMT Invoke recieve 3 parameters : p1=%d,p2=%d,p3=%d 100 200 300
SMT invoke recieve 3 parameters: p1=100,p2=200,p3=300
invoke is running buyOrder
invokeParam1
a
invokeParam3
buyerName=,status=0,price=
(t *writeOrder) UnmarshalJSON()
Unmarshal order from stub.GetState as bytes:
[]
------------json.Marshal(order) start-----
buyerName=a,status=888,price=invokeParam3
(t *writeOrder) MarshalJSON()
[105 110 118 111 107 101 80 97 114 97 109 49]
---------------json.Marshal ok--------------success
------------buyOrder end-----------
[105 110 118 111 107 101 80 97 114 97 109 49]
bytes -------------test from GetState
(t *writeOrder) UnmarshalJSON()
Unmarshal--------test from GetState
buyerName=a,status=888,price=invokeParam3
None
------------超级账本订单Order形式化验证----end------
State 0 :: [1]
2021-07-04 21:11:40,201: [4285] m.c.manticore:INFO: Generated testcase No. 23 - Execution returned 1
2021-07-04 21:11:40,573: [4285] m.c.manticore:INFO: Results in ./work/workspace
INFO:     127.0.0.1:52976 - "POST /chaincode/verify/go/ HTTP/1.1" 200 OK
```

Figure 17: A Hyperledger Fabric chaincode source file and symbolic execution in verification service

After symbolic execution in Fig. 17, many test cases will be generated in the server workspace. Finally, the verification service will automatically generate a detection report. In Fig. 18, a web HTML demonstrates a result of one detection report of a vulnerability function that has an expression: $1/(8 - x)$.



Detection Report

State0:: Input symbolic Value =[2]

State1:: Input symbolic Value =[3]

State2:: Input symbolic Value =[8]

Conclusion:

vulnerability test case : Input Value = 8

cause of vulnerability : The dividend is Zero.

Figure 18: Test cases of symbolic execution and a simple detection reported by FVPS

4 Conclusions

This work constructs a whole integrated WebAssembly formal verification platform. Furthermore, it applies to the Hyperledger Fabric blockchain platform and achieves great optimization in the golang language. We conclude that the compiler appends extra housekeeping codes in the user's source codes to complete the compilation. Besides, different languages have a corresponding amount number of extra codes. Although the compilation result has a relative highlighted effect in C/C++ in Fig. 7, the total size is too small, less than 20 KB. Hence the previous papers ignore code inflation affection of the housekeeping code. Our experiments show that program languages like golang severely impact code inflation. That is to say, if one program language compiler includes a runtime in housekeeping codes, our approach can gain a remarkable effect. FVPS reduces the code size by over 90%, from 2000 to 90 KB. That means FVPS can cope with golang applications up to 20 times larger than the original in scale. FVPS attempts to justify WebAssembly's viability as a smart contract language, and we eliminate the gap between Hyperledger Fabric smart contracts and WebAssembly. All of the related work can benefit from this touchstone work. Due to the massive workloads, the main shortcoming of this work is the lack of complex smart contract check models in formal verification work, and this is also the following work for us.

Funding Statement: This work was supported by the National Key R&D Program of China, Grant No. 2018YFA0306703.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] S. M. Jain, *WebAssembly introduction*. Berkeley, CA, USA: Apress, 2021. [Online]. Available at: https://doi.org/10.1007/978-1-4842-7496-5_1.

[2] M. Jacobsson and J. Willén, *Virtual machine execution for wearables based on WebAssembly*. Cham, Switzerland: Springer, 2020. [Online]. Available at: https://doi.org/10.1007/978-3-030-29897-5_33.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman *et al.,* "Bringing the web up to speed with WebAssembly," in *Proc. PLDI*, Barcelona, Spain, pp. 185–200, 2017.

[4] T. Gayvoronskaya and C. Meinel, *Blockchain*. Cham, Switzerland: Springer, 2020. [Online]. Available at: https://doi.org/10.1007/978-3-030-61559-8.

[5] E. Elrom, *EOS.IO Wallets and Smart Contracts*. Berkeley, CA, USA: Apress, 2019. [Online]. Available at: https://doi.org/10.1007/978-1-4842-4847-8_6.

[6] S. A. Bragadeesh and A. Umamakeswari, "Secured vehicle life cycle tracking using blockchain and smart contract," *Computer Systems Science and Engineering*, vol. 41, no. 1, pp. 1–18, 2022.

[7] Y. Ren, Y. Leng, J. Qi, S. K. Pradip, J. Wang *et al.,* "Multiple cloud storage mechanism based on blockchain in smart homes," *Future Generation Computer Systems*, vol. 115, no. 2, pp. 304–313, 2021.

[8] Y. Ren, F. Zhu, K. S. Pradip, T. Wang, J. Wang *et al.,* "Data query mechanism based on hash computing power of blockchain in internet of things," *Sensors*, vol. 20, no. 1, pp. 1–22, 2020.

[9] Z. Shahbazi and Y. Byun, "Blockchain and machine learning for intelligent multiple factor-based ride-hailing services," *Computers, Materials & Continua*, vol. 70, no. 3, pp. 4429–4446, 2022.

[10] S. R. Khonde and V. Ulagamuthalvi, "Blockchain: Secured solution for signature transfer in distributed intrusion detection system," *Computer Systems Science and Engineering*, vol. 40, no. 1, pp. 37–51, 2022.

[11] W. Cao, Z. Xie, X. Zhou, X. Z., C. Zhou *et al.,* "A learning framework for intelligent selection of software verification algorithms," *Journal on Artificial Intelligence*, vol. 2, no. 4, pp. 177–187, 2020.

[12] C. A. R. Hoare, J. Misra, G. T. Leavens and N. Shankar, "The verified software initiative: A manifesto," *ACM ComputingSurveys*, vol. 41, no. 4, pp. 1–8, 2009.

[13] I. Grishchenko, M. Maffei and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Proc. POST*, Thessaloniki, Greece, pp. 243–269, 2018.

[14] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian *et al.,* "Kevm: A complete formal semantics of the ethereum virtual machine," in *Proc. CSF*, Oxford, UK, pp. 204–217, 2018.

[15] S. Amani, M. Bégel, M. Bortin and M. Staples, "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proc. CPP*, Los Angeles, CA, USA, pp. 66–77, 2018.

[16] L. Luu, D. H. Chu, H. Olickel, P. Saxena and A. Hobor, "Making smart contracts smarter," in *Proc. CCS*, Vienna, Austria, pp. 254–269, 2016.

[17] Z. Wei, J. Wang, X. Shen and Q. Luo, "Smart contract fuzzing based on taint analysis and genetic algorithms," *Journal of Information Hiding and Privacy Protection*, vol. 2, no. 1, pp. 35–45, 2020.

[18] J. X. He, M. Balunovic, N. Ambroladze, P. Tsankov and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. CCS*, London, ENGLAND, pp. 531–548, 2019.

[19] W. Wang, J. Song, G. Xu, Y. Li and H. Wang, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko *et al.,* "Smartcheck: Static analysis of ethereum smart contracts," in *Proc. WETSEB*, Gothenburg, Sweden, pp. 9–16, 2018.

[21] Z. Yang and H. Lei, "A formal process virtual machine for EOS-based smart contract security verification," in *Proc. FICC*, Singapore, Springer, pp. 253–263, 2021.

[22] Z. H. Yan, W. Z. Qian, Z. Yang, W. R. Zeng, X. Yang *et al.,* "Tffv: Translator from EOS smart contracts to formal verification language," in *Proc. ICAIS*, Hohhot, China, pp. 652–663, 2020.

[23] J. T. George, *Hyperledger fabric*. Berkeley, CA, USA: Apress, 2021. [Online]. Available at: https://doi.org/10.1007/978-1-4842-7480-4_6.

[24] M. Uddin, M. S. Memon, I. Memon, I. Ali, J. Memon *et al.,* "Hyperledger fabric blockchain: Secure and efficient solution for electronic health records," *Computers, Materials & Continua*, vol. 68, no. 2, pp. 2377–2397, 2021.

[25] X. R. Zhang, X. Sun, X. M. Sun, W. Sun and S. K. Jha, "Robust reversible audio watermarking scheme for telemedicine and privacy protection," *Computers, Materials & Continua*, vol. 71, no. 2, pp. 3035–3050, 2022.