Specification of DSL for Hyperledger Fabric Chaincode

Ningyu He

May 2021

1 Term

Our self-defined DSL contains four kinds of terms: integer constants, identifiers, qualifiers and operators.

2 Syntax

E is an expression, defined as:

$$\begin{aligned} E &\rightarrow intconst \\ &\rightarrow id \\ &\rightarrow E_1 \texttt{ bop } E_2 \\ &\rightarrow \texttt{uop } E \\ &\rightarrow E_1 \implies E_2 \\ &\rightarrow (E) \end{aligned}$$

Specifically, 1) `bop` refers to binary operators, including arithmetic operators (e.g., `+` and `-`), comparison operators (e.g., `>` and `!=`), logical operators (e.g., `and` and `or`) and a special `$` that will be expressed later; 2) `uop` refers to unary operators, such as negative operators; 3) *intconst* refers to integer constant; 4) *id* refers to identifiers; and 5) $\implies$ refers to the 'imply' relationship, its value is equivalent to $\neg E_1 \vee E_2$. Note that, we interpret 0 as $False$ and others as $True$.

Further, we define a state π as:

$$\pi := \bigcap id \mapsto \alpha$$

, which is the intersection of the mapping that is from variable to its integer constant or a symbol at a given point in the program.

Based on the expression, the statement S could be defined as follows:

$$\begin{aligned} S &\rightarrow \texttt{@assume} : E_{pre} \\ &\rightarrow \texttt{@ensure} : E_{post} \\ &\rightarrow \texttt{@require} : E_{require} \\ &\rightarrow \texttt{@forall} : id \texttt{ \$ } E_{forall} \end{aligned}$$

To be specific, `assume` and `ensure` are predicates that should hold upon entry into and exit from the function, respectively. The `forall` will iterate the variable *id* and guarantee all its elements holds for E. At last, `require` can be inserted during the function arbitrarily to check if the E holds at the given point.

3 Semantic

To further introduce the semantics of our DSL, firstly we formally represent the function that will be symbolical executed as follows:

$$f := C_1; C_2; ...; C_n$$

, where C_i can be seen as a code snippet, and the function is spliced together of those snippets.

Moreover, we borrow the concept of Hoare Logic, like:

$$\{P\}f\{Q\}$$

, where P and Q are the precondition and postcondition of function f, respectively. Moreover, if for all Q such that $\{P\}f\{Q\}$, $Q \implies Q$, the Q is called *the strongest postcondition*. In the following, we will omit this condition, and assume all the Qs are the strongest ones.

Based on the above formal representation, we could formally define S mentioned in §2.

Assume

$$\frac{}{E_{pre}} \tag{1}$$

@`assume` : E_{pre} must be declared before the function's body, and it indicates that E_{pre} should unditionally hold upon entry into the function, or the function will not be analyzed any more.

Ensure

$$\frac{E_{pre} \wedge \{\pi_{pre}\}f\{\pi_{post}\}}{\pi_{post} \implies E_{post}} \tag{2}$$

Similarly to the `@assume`, the `@ensure` should also be declared before the function's body. Specifically, `@ensure` will guarantee: 1) the precondition E_{pre} holds; and 2) after executing the whole program, the strongest postcondition π_{post} will imply the E_{post}.

Require

$$\frac{E_{pre} \wedge \{\pi_{pre}\}C_1; ...; C_i\{\pi'\}}{\pi' \implies E_{require}}, \text{where } 1 \leq i \leq n, i \in \mathbb{N}^* \tag{3}$$

As with the semantic of `@ensure`, the `@require` also guarantees two predicates: the precondition E_{pre} holds, and the strongest postcondition could imply the $E_{require}$. However, the position of `@require` is more flexible, thus the strongest postcondition π' is obtained after executing *a piece of program* instead of the whole program.

Forall

$$\frac{E_{pre} \wedge \{\pi_{pre}\}C_1; ...; C_i\{\pi'\}}{\forall v \in id.(\pi' \implies E_{forall}[\![v]\!])}, \text{where } 1 \leq i \leq n, i \in \mathbb{N}^* \tag{4}$$

The `@forall` is to facilitate the analysis of a certain type of variable: *iterable variable*. To be specific, it can be regarded as an enhanced version of `@require`. The `@forall` will iterate the *id* to verify if all the predicates E_{forall}, restricted to element v, can be implied by the strongest postcondition π'.

All the above four types of statement can be utilized by the symbolic execution engine to prune unfeasible paths in advance to accelerate the analysis process.

4 Example of Statements

With the help of this four types of statement, we could easily verify if the given piece of code violate the given patterns. For example, listing 1 shows the original piece of code, which performs a simple arithmetic operation in Go.

```
1  package main
2
3  import (
4    "fmt"
5    "math/rand"
6  )
7
8  func testPre(a uint8) uint8 {
9    a *= 10
10   return a
11 }
12
13 func genRans() []int {
14   var rans []int
15   for i := 0; i < 5; i++ {
16     rans = append(rans, rand.Intn(20))
17   }
18   return rans
19 }
20
21 func main() {
22   var a, b uint8
23   a = 25
24   a = testPre(a)
25   b = 10
26   d := genRans()
27   fmt.Println(d)
28
29   c := a + b
30   fmt.Printf("c is: %v, type is: %T\n", c, c)
31 }
```

Listing 1: Original piece of code

By adding some statement according to our definitions of DSL, the original code could be like listing 2.

```
1  package main
2
3  import (
4    "fmt"
5    "math/rand"
6  )
7
8  // @assume: a <= 25;
9  func testPre(a uint8) uint8 {
10   a *= 10
```

```
  return a
}

func genRans() []int {
  var rans []int
  for i := 0; i < 5; i++ {
    rans = append(rans, rand.Intn(20))
  }
  return rans
}

// @ensure: a + b > a;
func main() {
  var a, b uint8
  a = 25
  a = testPre(a)
  b = 10
  d := genRans()
  fmt.Println(d)
  // @forall: d $ _el <= 20

  c := a + b
  fmt.Printf("c is: %v, type is: %T\n", c, c)
  // @require: c > a && c > b;
}
```

Listing 2: Commented code

Therefore, the parser will parse the comments and insert some codes, i.e., assertions, in the original code, which would be like listing 3.

```
package main

import (
  "fmt"
  "math/rand"
)

func testPre(a uint8) uint8 {
  if !(a <= 25) {
    panic("against @assume: a <= 25")
  }
  a *= 10
  return a
}

func genRans() []int {
  var rans []int
  for i := 0; i < 5; i++ {
    rans = append(rans, rand.Intn(20))
  }
  return rans
}

func main() {
  var a, b uint8
  a = 25
  a = testPre(a)
  b = 10
  d := genRans()
```

```
  fmt.Println(d)
  for _, _el := range d {
    if !(_el <= 20) {
      panic("against @forall: d $ _el <= 20")
    }
  }

  c := a + b
  fmt.Printf("c is: %v, type is: %T\n", c, c)
  if !(c > a && c > b) {
    panic("against @require: c > a && c > b")
  }

  if !(a+b > a) {
    panic("against @ensure: a + b > a")
  }
}
```

Listing 3: Parsed code

At last, by symbolically executed the parsed code, the paths that satisfy all the predicates will be kept and the corresponding constraints will be recorded.

5 Predefined Security Check

Except for the four types of statements mentioned in §2, we provide some higher-order statements for users to verify some general vulnerabilities as following:

$$\begin{aligned} S &\rightarrow \texttt{@check} : \texttt{integerOverflow} \\ &\rightarrow \texttt{@check} : \texttt{bufferOverflow} \\ &\rightarrow \texttt{@check} : \texttt{nullPointerDereference} \end{aligned}$$

Specifically, the position of `@check` is as with the `@assume` and `@ensure`, i.e., declaring before the function to be verified. For example, Listing 4 illustrates how to use the `@check` statement.

```
package main

// @check: integerOverflow
func main() {
  // some operations
}
```

Listing 4: An example of using `@check` statement

5.1 Verify integer overflow vulnerability

Integer overflow in Ethereum smart contracts is studied by [3]. And it is solely introduced by the arithmetic operations, including `add`, `sub`, `mul` and so on. However, except for the variety of arithmetic operations, there are several basic types for integer, like `uint8`, `int16`. Therefore, it is costly to go through all combinations and insert corresponding assertions (like the way we did in `@require`) for them to check for integer overflow. To this end, we decide to implement the function-level integer overflow verification in our symbolic execution engine. Specifically, for each of arithmetic operations, we would insert an additional constraint to examine if the result is overflowed.

5.2 Verify buffer overflow vulnerability

The buffer overflow vulnerability in WebAssembly is well studied in [2]. To be specific, there are two types of variables in WebAssembly modules: *maintained data* and *unmaintained data*. For the former one, e.g., `local` and `global`, it can only be interacted with through WebAssembly's instructions; as for the later one, it is completely stored in the linear memory area, which is unprotected by any of the memory protection mechanisms, e.g., unmapped pages, page protection flags and address space layout randomization (ASLR). Moreover, [2] also proposed a heuristic methodology to determine the size of the current stack frame δ. Therefore, to scan the buffer overflow vulnerability, our symbolic execution engine would try to insert a piece of data, at least longer than δ, if it is possible to write something into the stack.

5.3 Verify null pointer dereference vulnerability

A pointer in WebAssembly will be uniformly defined as an `uint32` variable. With the help of our framework's memory emulation [1], a null pointer would be easily observed by if the loaded area is initialized before. Therefore, if user tends to dereference a null pointer, the destination of `load` instruction would be a vacuum, which can be easily captured by our framework.

References

[1] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. {EOSAFE}: Security analysis of {EOSIO} smart contracts. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[2] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 217–234, 2020.

[3] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 664–676, 2018.