Lab: Z3 Theorem Prover

(Week 4)

Yulei Sui, Mohamad Barbar, Hanyuan Li, Angela He

School of Computer Science and Engineering

University of New South Wales, Australia

Quiz-2 and Exercise-2

Remember to `git pull` or `docker pull`!

Please update your code template to the latest version for Lab-Exercise-2.

Quiz-2 and Exercise-2

- Quiz-2 with 25 questions (5 points), **due date: 23:59 Wednesday, Week 7**
  - Logical formula and predicate logic
  - Z3's knowledge and translation rules
- Lab-Exercise-2 (5 points), **due date: 23:59 Wednesday, Week 7**
  - **Goal:** Manually translate code into z3 formulas/constraints and verify the assertions embedded in the code.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Lab-Exercise-2`
  - **SVF Z3 APIs:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-Z3-API`
- Assignment-2 (25 points) will **start from Week 5 and due date: 23:59 Wednesday, Week 8**
  - **Goal:** automatically perform assertion-based verification for code using static symbolic execution.
  - **Specification:** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-2`

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).
- Predicate logic adds **quantifiers** ($\forall$, $\exists$) to **propositional logic**, allowing us to *encode functions* and *relate statements with each other*.

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).
- Predicate logic adds **quantifiers** ($\forall$, $\exists$) to **propositional logic**, allowing us to *encode functions* and *relate statements with each other*.
  - $\{\forall x : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).
- Predicate logic adds **quantifiers** ($\forall$, $\exists$) to **propositional logic**, allowing us to *encode functions* and *relate statements with each other*.
  - $\{\forall x : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$
  - $\{P(x) : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$

Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).
- Predicate logic adds **quantifiers** ($\forall$, $\exists$) to **propositional logic**, allowing us to *encode functions* and *relate statements with each other*.
  - $\{\forall x : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$
  - $\{P(x) : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$
  - $\{\varphi; x \geq 20; \texttt{if}(x \geq 10)\{y = 5\}\} \vdash y = 5$ (**constrained Horn clause**)

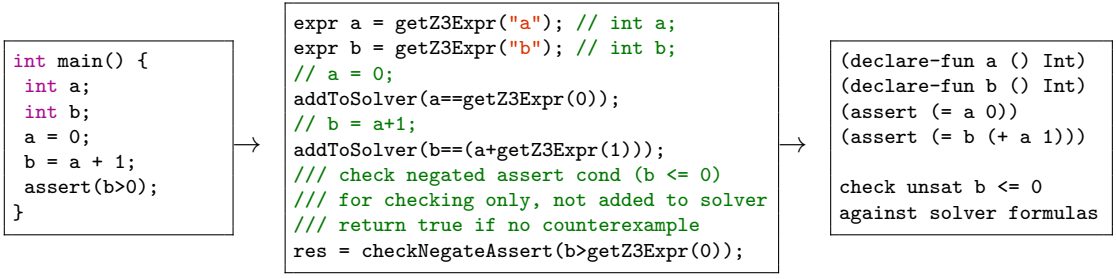
Z3 recap

Z3 is an **automated satisfiable modulo theory (SMT) solver** that can accept **predicate logic** formulas.

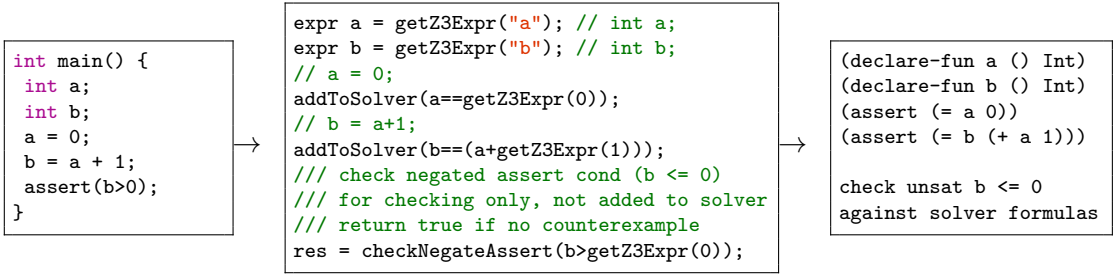
- SMT is a generalisation of the **boolean satisfiability problem (SAT)** to include *integers* and more complex data structures (e.g. *lists*, *strings*).
- Predicate logic adds **quantifiers** ($\forall$, $\exists$) to **propositional logic**, allowing us to *encode functions* and *relate statements with each other*.
  - $\{\forall x : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$
  - $\{P(x) : x \geq 85 \implies x \text{ has HD}; \text{John} \geq 85\} \vdash \text{John has HD}$
  - $\{\varphi; x \geq 20; \texttt{if}(x \geq 10)\{y = 5\}\} \vdash y = 5$ (**constrained Horn clause**)

Predicate logic allows us to encode **programs** (series of related statements) and **functions** (quantified statements), SMT allows us to work with **higher-level data types**.

Z3 recap


int main() {
int a;
int b;
a = 0;
b = a + 1;
assert(b>0);
}
expr a = getZ3Expr("a"); // int a;
expr b = getZ3Expr("b"); // int b;
// a = 0;
addToSolver(a==getZ3Expr(0));
// b = a+1;
addToSolver(b==(a+getZ3Expr(1)));
/// check negated assert cond (b <= 0)
/// for checking only, not added to solver
/// return true if no counterexample
res = checkNegateAssert(b>getZ3Expr(0));
(declare-fun a () Int)
(declare-fun b () Int)
(assert (= a 0))
(assert (= b (+ a 1)))
check unsat b <= 0
against solver formulas

Z3 recap



Using Z3, we can automatically find bugs in our code!

Learning resources

Extra propositional/predicate logic slides from last year:
`https://webcms3.cse.unsw.edu.au/COMP6131/26T2/resources/123732`

Learning resources

Extra propositional/predicate logic slides from last year:
`https://webcms3.cse.unsw.edu.au/COMP6131/26T2/resources/123732`

Further courses on this topic:

- COMP4141: Theory of Computation (SAT, P-NP problem, complexity)
- COMP4161: Advanced Topics in Software Verification (proof engines)

Z3 API

Z3Examples API	Meanings
`z3::expr getZ3Expr(std::string);`	Create a z3 expr given a string name
`z3::expr getZ3Expr(u32_t);`	Create a z3 expr given an integer
`z3::expr getCtx().int_val(u32_t);`	Create a z3 expr given an integer
`z3::expr getMemObjAddress(std::string);`	Create a **memory object** in program
`z3::expr getGepObjAddress(z3::expr, u32_t);`	Create a **field object** with an **offset** of an aggregate
`void addToSolver(z3::expr);`	**Add** a Z3 expression/formula to the solver
`void resetSolver();`	Clean all formulas in the the solver
`check_result solver.check();`	Check if a formula is satisfiable; return sat/unsat/unknown
`bool checkNegateAssert(Z3Mgr, z3::expr);`	**Check negated** assert return true if no counterexample
`z3::expr getEvalExpr(z3::expr);`	**Evaluate an expression** to a value based on a model
`void printExprValues();`	**Print** the values of **all expressions** in the solver
`void printZ3Exprs();`	**Print** all **z3 formulas** in the solver

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1 expr p = getZ3Expr("p");
2 expr q = getZ3Expr("q");
3 expr r = getZ3Expr("r");
4 expr x = getZ3Expr("x");
5 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----


----------------------------
```

nothing printed because expressions have no value

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1 expr p = getZ3Expr("p");
2 expr q = getZ3Expr("q");
3 expr r = getZ3Expr("r");
4 expr x = getZ3Expr("x");
5 expr malloc1 = getMemObjAddress("malloc1");
6 addToSolver(p == malloc1);
7 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
----------------------------
```

`0x7f000005` (or 2130706437 in decimal) represents the virtual memory address of this object

Each `ObjVar` starts with `0x7f` + its ID.

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5
----------------------------
```

store value of `q` to address `0x7f000005`

load the value from `0x7f000005` to `x`

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
11
12 // use checkNegateAssert to unsat of x!=10
13 // could check sat of x==10 due to
14 // closed-world program
```

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5

Assertion failed: (false &&
"The assertion is unsatisfiable");
---------------------------
```

Contradictory Z3 constraints!

$x \equiv 5$ contradicts $x \equiv 10$

Source code

Translation code using Z3Mgr

Output on terminal

Intraprocedural Example

```
1 int* p;
2 int q;
3 int* r;
4 int x;
5 p = malloc(...);
6 q = 5;
7 *p = q;
8 x = *p;
9 assert(x==10);
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr x = getZ3Expr("x");
5  expr malloc1 = getMemObjAddress("malloc1");
6  addToSolver(p == malloc1);
7  addToSolver(q == getZ3Expr(5));
8  storeValue(p, q);
9  addToSolver(x == loadValue(p));
10 printExprValues();
11
12 /// evaluation code as below
13 std::cout<< getEvalExpr(x == getZ3Expr(10))
14 << std::endl;
```

```
----------Var and Value-----
Var5 (malloc1)    Value: 0x7f000005
Var1 (p)          Value: 0x7f000005
Var2 (q)          Value: 5
Var4 (x)          Value: 5
false
----------------------------
```

There is no model available (unsat)

when evaluating `x == getZ3Expr(10)`

Source code

Translation code using Z3Mgr

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  solver.check();
7  expr r = getEvalExpr(a);
8  printExprValues();
9  solver.pop();
10 addToSolver(p == r);
```

Handle first callsite `p=bar(2)`

Translation code using Z3Mgr

```
----------Var and Value-----
Var2 (a)          Value: 2
-----------------------------
```

(1) `push` the z3 constraints when calling `bar` and `pop` when returning from `bar`

(2) Expression `r` is the return value evaluated from `a` after returning from callee `bar`

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  solver.check();
7  expr r = getEvalExpr(a);
8  solver.pop();
9  addToSolver(p == r);
10 printExprValues();
```

Handle first callsite `p=bar(2)`

```
----------Var and Value-----
Var1 (p)        Value: 2
----------------------------
```

Now we only have `p`'s value and `a` is not in the current stack since constraint `a == getZ3Expr(2)` has been popped

Source code

Translation code using Z3Mgr

Output on terminal

Interprocedural Example (Call and Return)

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  solver.push();
4  expr a = getZ3Expr("a");
5  addToSolver(a == getZ3Expr(2));
6  expr r = getEvalExpr(a);
7  solver.pop();
8  addToSolver(p == r);
9  solver.push();
10 addToSolver(a == getZ3Expr(3));
11 r = getEvalExpr(a);
12 solver.pop();
13 addToSolver(q == r);
14 printExprValues();
```

Handle second callsite `q=bar(3)`

```
----------Var and Value-----
Var1 (p)         Value: 2
Var2 (q)         Value: 3
----------------------------
```

We have two expressions and their values in main's scope

Source code | Translation code using Z3Mgr | Output on terminal

Bad Interprocedural Example Without `push/pop`

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr a = getZ3Expr("a");
4  addToSolver(a == getZ3Expr(2));
5  expr r = getEvalExpr(a);
6  addToSolver(p == r);
7  addToSolver(a == getZ3Expr(3));
8  r = getEvalExpr(a);
9  addToSolver(q == r);
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Assertion failed: (res!=z3::unsat &&
"unsatisfied constraints! Check your
contradictory constraints added to
the solver")
----------------------------
```

Output on terminal

both `a == getZ3Expr(2)` and `a == getZ3Expr(3)` are added into the solver in the same scope

Bad Interprocedural Example Without Evaluating Return

```
1  int bar(int a){
2      int r = a;
3      return r;
4  }
5  void main(){
6      int p, q;
7      p = bar(2);
8      q = bar(3);
9      assert(p==2)
10 }
```

Source code

```
1  expr p = getZ3Expr("p");
2  expr q = getZ3Expr("q");
3  expr r = getZ3Expr("r");
4  expr a = getZ3Expr("a");
5  solver.push();
6  addToSolver(a == getZ3Expr(2));
7  addToSolver(r == a);  // invalid after pop
8  solver.pop();
9  addToSolver(p == r);
10 printExprValues();
11 solver.push();
12 addToSolver(a == getZ3Expr(3));
13 addToSolver(r == a);  // invalid after pop
14 solver.pop();
15 addToSolver(q == r);
16 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (p)         Value: random
Var2 (q)         Value: random
Var3 (r)         Value: random
----------------------------
```

Output on terminal

the values of `p,q,r` are

the same random number

Array and Struct Example

```
1  void main(){
2      int* a;
3      int* x;
4      int y;
5      a = malloc(...);
6      x = &a[2];
7      *x = 3;
8      y = *x;
9      assert(y==3);
10 }
```

Source code

```
1  expr a = getZ3Expr("a");
2  expr x = getZ3Expr("x");
3  expr y = getZ3Expr("y");
4  addToSolver(a == getMemObjAddress("malloc"));
5  addToSolver(x == getGepObjAddress(a,2));
6  storeValue(x, getZ3Expr(3));
7  addToSolver(y == loadValue(x));
8
9  /// print expr values as below
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (a)          Value: 0x7f000004
Var4 (malloc)     Value: 0x7f000004
Var2 (x)          Value: 0x7f000003
Var3 (y)          Value: 3
----------------------------
```

`getGepObjAddress` returns the field address of the aggregate object *a*

The virual address also in the form of `0x7f.. + VarID`

Output on terminal

Array and Struct Example

```
1  void main(){
2      int* a;
3      int* x;
4      int y;
5      a = malloc(...);
6  // similar for struct
7  //   x=&(a->fld2)
8      x = &a[2];
9      *x = 3;
10     y = *x;
11     assert(y==3);
12 }
```

Source code

```
1  expr a = getZ3Expr("a");
2  expr x = getZ3Expr("x");
3  expr y = getZ3Expr("y");
4  addToSolver(a == getMemObjAddress("malloc"));
5  addToSolver(x == getGepObjAddress(a,2));
6  storeValue(x, getZ3Expr(3));
7  addToSolver(y == loadValue(x));
8
9  /// print expr values as below
10 printExprValues();
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (a)          Value: 0x7f000004
Var4 (malloc)     Value: 0x7f000004
Var2 (x)          Value: 0x7f0001f7
Var3 (y)          Value: 3
----------------------------
```

`getEvalExpr` retrieve the value from the expression

Output on terminal

Branch Example

```
1  void main(int argv){
2      int y = 2;
3      if(argv > 2)
4          y = argv;
5
6  // both definitions of y
7  // at lines 2 and 4 go to
8  // this joint point
9      assert(y>=2);
10 }
```

Source code

```
1  expr argv = getZ3Expr("argv");
2  expr y = getZ3Expr("y");
3  // new variable y1 to mimic the phi to merge
4  // two definitions of y at control flow joint point
5  expr y1 = getZ3Expr("y1");
6  expr two = getZ3Expr(2);
7  addToSolver(y == two);
8  addToSolver(y1 == ite(argv > two, argv, y));
9
10 /// expr validation and evaluation as below
11 printExprValues();
12 std::cout<<getEvalExpr(y1 >= two)<<"\n";
```

Translation code using Z3Mgr

```
----------Var and Value-----
Var1 (argv)          Value: 0
Var2 (y)             Value: 2
Var3 (y1)            Value: 2
----------------------------
true
```

Output on terminal

Quiz-1 and Exercise-1 Marks Released

- Marks are out and check them on `give`:
  `https://cgi.cse.unsw.edu.au/~give/code/login.php`.
- We will go through Quiz-1 questions and common issues of Exercise-1.