Efficient and effective symbolic execution: technologies overview

Sergey Morozov
morozov.serg901@gmail.com

Aleksandr Misonizhnik
misonijnik@gmail.com

June 28, 2023

Automatic Program Analysis

We want a tool, that will be able to find vulnerabilities in a given source code

Approaches to Program Analysis

- Abstract interpretation is useful for verifying safeness, but not always be efficient
- Fuzzing is very good at finding edge cases, but it can miss certain types of bugs
- ...
- **There are very fast static analysis approaches, but not very accurate**

Symbolic Execution Approach

- Symbolic execution is a very precise technique, but accuracy comes at the cost of time
- Core idea — explore all possible program behaviours

Symbolic Execution: General ideas

- Introduces a symbolic variable
- Constructs logical formulas
- Checks satisfiability with SMT-solver
- For each state achieved endpoint generates a set of values for symbolic variables

```
int main() {
    int x = symbolic();
    if (x > 0) {
        return 0;
    }
    return 1;
}
```

Classic Symbolic Execution: Algorithm

```
Q_front := {s};
while Q_front ≠ ∅ do
    s := searcher.pick(Q_front);
    Q_front := Q_front \ {s};
    forall s' ∈ execInstr(s.curr, s) do
        if isSAT(s'.pc) and checkBound(s')
            Q_front := Q_front ∪ {s'};
```

- Key points:
  - path selection heuristic (`searcher.pick`)
  - program execution modelling (`execInstr`)
  - logic solver (`isSAT`)
- Improvements can be made in each of the key points to achieve acceptable performance for production use

Path Selection

Q_{front} := $\{s\}$;
while $Q_{front} \neq \varnothing$ **do**
  s := `searcher.pick(`Q_{front}`);`
  Q_{front} := $Q_{front} \setminus \{s\}$;
  forall $s' \in$ `execInstr(`$s.curr$, s`)` **do**
    if `isSAT(`$s'.pc$`)` **and** `checkBound(`s'`)`
      Q_{front} := $Q_{front} \cup \{s'\}$;

Responsible for choosing state for execution
Each state can be represented as a vertex in Control Flow Graph
To traverse graph efficiently engine may use different algorithms:

- DFS
- BFS
- Random Walk
- Weighted Random Walk
- ...

Path Selection

- Affects completeness of analysis
- Not every algorithm can be used effectively

Paths Selection: Does it work?

In most cases. But not well:

```
int foo(int x) {
  int y = 0;
  for (int i = 0; i < x; ++i, ++y) {}
  if (y == 250) { printf("y == 250");}
  return y;
}
```

Execution with any searcher stucks in `for`-loops

Path Selection: Guided mode

- But what if we try to analyse entire graph?
- We could **guide** execution to the interesting **targets**

Path Selection: Guided Mode

Introduce a **guided searcher** that manages many **targeted searchers**

- Each targeted searcher manages its own set of states and choose states that will *likely* achieve target
- By default states *does not* have targets
- But if state without target passes same instruction many times, guided searcher will **calculate** a target to it
- If target has been reached or target can not be reached, then state loses target

Path Selection: Targeted Searchers

Manages set of states with the same target

- If target is in the same function, calculates distance as number of instructions
- Otherwise, calculates shortest path in the call graph with transitions on `call`'s and `return`'s

Path selection: it works faster!

```c
int foo(int x) {
  int y = 0;
  for (int i = 0; i < x; ++i, ++y) {}
  if (y == 250) { printf("y == 250");}
  return y;
}
```

With **Guided Mode** it works almost 5 time faster!

Path Selection: open problem

```
void f(int n, int k) {
  while (true) {
    if (n == 50 && k <= 50)
      return;
  }
}

void g(int n, int k) {
  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
      if (n == i && k == j)
        f(n, k);
    }
  }
}
```

- However, does not work always
- Suppose, we want to analyse function g
- With our approach it will hang

Memory Model

```
Q_front := {s};
while Q_front ≠ ∅ do
    s := searcher.pick(Q_front);
    Q_front := Q_front \ {s};
    forall s′ ∈ execInstr(s.curr, s) do
        if isSAT(s′.pc) and checkBound(s′)
            Q_front := Q_front ∪ {s′};
```

- Important part for analysis for any language with dynamic memory allocations
- One of the sources of developers errors
  - Out Of Bound
  - Null Pointer Dereferences
  - Uses After Free
  - ...
- Therefore we need to maintain **correct** memory representation for each stat

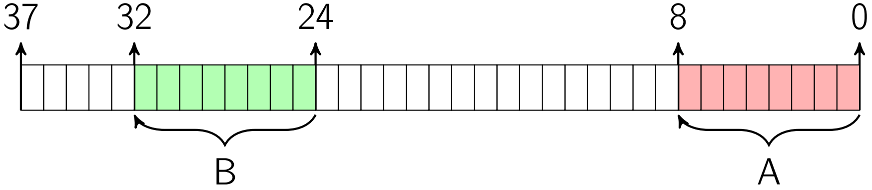
Memory Model

```
bool buf[CHAR_BIT];
for (int i = 0; i <= CHAR_BIT; ++i) {
  buf[i] = 0;
}
```

We may see a typical out-of-bound error on line `buf[i] = 0;`

Memory Model:

Address space in a program may be represented as a contiguous segment with objects inside



The main operation in memory is a **pointer resolution**. It maybe either:

- Concrete
- Symbolic

Memory Model: Pointer Resolution

- Resolution of symbolic pointer looks over **all objects** in memory which can be referred by it and forks initial state with reads and writes in that objects
- We may see at least 2 problems:
    - Performance
    - Completeness

Pointer Resolution: Performance

```
int main() {
  float a;
  short b;
  int d;

  int *x = symbolic();
  *x = 10;
}
```

As pointer can be dereferred in every object, we will create lots of additional execution states

Pointer Resolution: Why is it that slow?

- In example before we have dereffered `int*` to `float`'s
- Do we really want to explore such behaviours?
- Idea: what if we restrict resolution with **type information**?

Memory Model: Type Information

- It is *language specific* information
- In C we have **Strict Aliasing Rule** and concept of objects **Effective Type**

Memory Model: Type Information

During pointer resolution we may also compare types of objects with the type of pointer in order to filter non-suitable ones

```
int main() {
  float a;
  short b;
  int d;

  int *x = symbolic();
  *x = 10;
}
```

With type system this example works almost 2 times faster

Memory Model: Completeness

What if we want to analyze recursive data structures?

```
struct Node {
  struct Node *next;
};
int len(struct Node *node) {
  if (node == NULL) {
    return 0;
  }
  return 1 + len(node->next);
}
int main() {
  struct Node node = symbolic();
  if (len(&node) > 1) { printf("len is %d 1!\n", len(&node)); }
}
```

Classic symbolic execution will generate lists with 1 nodes, may be with reference to itself

Memory Model: Lazy Initialization

Idea: allocate additional object!

- Even if the *symbolic* pointer points to nowhere, allocate an additional object with error report
- Symbolic pointer becomes address of allocated object
- Also we need to add logical constraints to prevent objects intersections in form of

$$base \leq ptr \land ptr + bytes \leq base + size$$

Memory Model: Lazy Initialization

It works!

```
struct Node {
  struct Node *next;
};
int len(struct Node *node) {
  if (node == NULL) {
    return 0;
  }
  return 1 + len(node->next);
}
int main() {
  struct Node node = symbolic();
  if (len(&node) > 1) { printf("len is %d 1!\n", len(&node)); }
}
```

Lazy Initialization: a problem

- To have correct memory model we must add constraints on non-intersections with every object for every lazy initialized object
- Otherwise we might receive memory model with different objects at one address
- Such constraints affect **performance**
- (Additionally) We do not know exact **size** of allocated object

To understand the problem, we will talk about **solvers**

Solvers: Satisfiability Modulo Theories

```
Q_front := {s};
while Q_front ≠ ∅ do
    s := searcher.pick(Q_front);
    Q_front := Q_front \ {s};
    forall s' ∈ execInstr(s.curr, s) do
        if isSAT(s'.pc) and checkBound(s')
            Q_front := Q_front ∪ {s'};
```

- SMT-solvers are widely used in symbolic execution
- Used to solve **logical formulas**
- Formulas consist of **logical theories**
  - BitVectors
  - Arrays
  - Linear Integer Arithmetic
  - ...

Solvers: SMT sounds good, right?

But

- SMT is an NP-hard problem
- Number and complexity of constraints affects solvers decising abilities
- Therefore, we should load the solver as little as possible

Lazy Initialization affects solvability

- Each lazy initialization adds conjunction of $O(n)$ constraints where n is the number of objects in the memory
- Problem — address space is a complex domain
- How we can even model complex domains?
- What if we make symbolics more *concrete*?

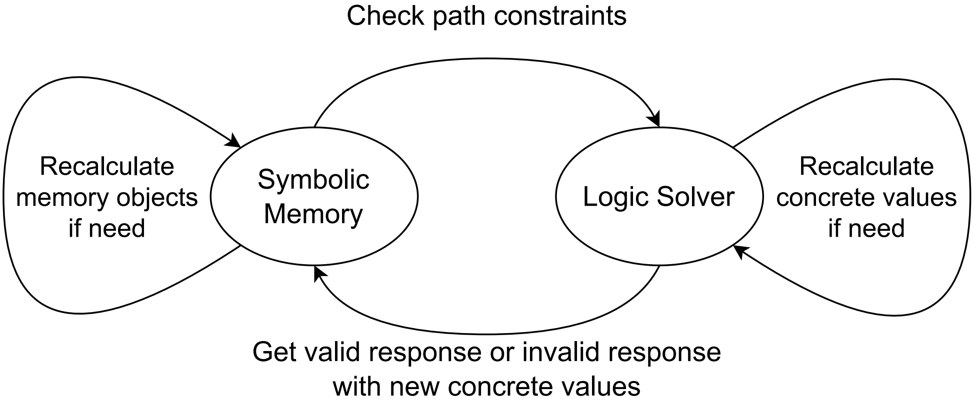
Symcretes infrastructure

We may delegate responsibility for generating values for symbolic variables to more light-weight algorithms then SMT-solvers. Therefore, we may use ideas of **symcrete execution**

- Symcrete = **sym**bolic + con**crete**
- Match symbolics with concrete values
- Allows to maintain a correct model

Symcretes overview

Symcretization approach: symcrete addresses and symcrete sizes



Symcretes machinery: solver

Managed with **Concretizing solver**

- Uses provided algorithms to generate solution for symcretes
- Modifies each query with equalities over symcrete variables
  - i.e. constraint $x < y$ with symcretes $(x = 2), (y = 1)$ will transform into $x < y \wedge x = 2 \wedge y = 1$
- Such modifications may affect formulas satisfiability. If so, remove all equalities over symcretes that affected validity
  - In example above we may remove $x = 2 \wedge y = 1$
- This is done by looking into **unsatisfiability core**

Symcretes machinery: memory

- With symcretes infrastructure we may control values of symbolic addresses with symcretes
- Moreover, we may maintain objects of symbolic size, as we are able to maintain correct model for all symbolic variable now
- “Solver” for addresses – allocator
  - `malloc(size_t)` function, for instance

Advantages of symcretes

- Simpler formulas for logic solver
- Most optimizations with objects of a concrete size continue to work with objects of symbolic size
- Try to keep the concrete sizes of the object as small as possible

Logic Solver: another optimizations and improvements

In order to work better with new functionality we've also made several optimizations:

- Use sparse storage for formula models
- Use interning of symbolic expressions to compare ones in constant time
- Support for queries to the solver to get an unsatisfiable core
- Cache all solver results to decrease time-consuming

Our Approach: KLEE-based implementation

- Our implementation is based on the KLEE symbolic execution engine

Current and Future Work

- Combining fuzzing and symbolic execution: using a fuzzy solver based on the libAFL fuzzer for exploring code with external function calls
- Combining static analysis and symbolic execution
- Combining reachability analysis and symbolic execution: bidirectional symbolic execution

Conclusion

- We have described approaches to automatic program analysis and, in particular, symbolic execution
- We have presented our approaches to make effectivity and effectiveness of symbolic execution, including
  - Guided mode for path selection
  - Type system and lazy initialization for work with memory
  - Symcretes approach to deal symbolic variables with complex domain
- Finally, we have discussed our ongoing and future work, including bidirectional symbolic execution, combining fuzzing and symbolic execution, and combining static analysis and symbolic execution

Articles

- MISONIZHNIK A. V. et al. Automated testing of LLVM programs with complex input data structures //Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS). – 2022. – T. 34. – №. 4. – C. 49-62.
- (To be published) MOROZOV S. A. et al. "Symcrete" memory model with lazy initialization and objects of symbolic sizes in KLEE (link)