Abstract Interpretation Meets Symbolic Execution

M2 Internship Proposal, Extended Version

1 Introduction

Context Abstract Interpretation [2, 3, 6] and Symbolic Execution [1, 4] are two program analysis techniques introduced in the mid-1970s that allow checking whether a program satisfies or violates certain properties. Abstract Interpretation approximates the semantics of a program using a monotonic function over ordered sets, called abstract domains, which allow for determining whether a given property is satisfied. Symbolic Execution, on the other hand, encodes the possible execution paths of a program as logical formulas and uses a constraint solver to generate inputs for executing feasible paths.

These two techniques were developed in different contexts for different purposes. Abstract Interpretation, as an over-approximation verification technique, ensures that a program contains no bugs but cannot find a bug if it fails to prove their absence. This technique converges quickly to a result and is primarily used in critical software certification. On the contrary, Symbolic Execution is a sub-approximation verification technique that can find actual bugs but cannot prove their absence unless infinite time is available. By its nature, Symbolic Execution cannot converge in finite time, but it has been successfully integrated into industrial contexts where it is common practice to leave programs under constant testing.

Problem Although complementary, these two techniques have developed largely independently, with little work focused on how to combine them. Some recent efforts, such as those in the Infer static analyzer [5, 8], have begun exploring this area, but they primarily focus on analyses integrated directly into the development process, which are limited to detecting so-called "surface" bugs. The goal of this project is therefore to explore the potential for cooperation between these techniques in uncovering "deep" bugs.

Proposal In this project, we plan to develop a static analyzer for the `While++` language. This analyzer will consist of an abstract interpreter, which is responsible for detecting potential bugs and inferring the necessary constraints to trigger them, and a symbolic executor, which will try to find a concrete path to the bug using these constraints. To achieve this, we will focus on the following aspects:

Numeric invariants We will begin with a textbook abstract interpreter designed to infer numeric invariants. When a bug is detected, we will employ backward reasoning to infer the numeric constraints necessary to trigger the bug. These constraints will then be passed to symbolic execution to avoid the unnecessary exploration of paths that cannot lead to the bug.

Initially, we will use a non-relational domain primarily because they are easier to implement. However, we anticipate that these domains may not be precise enough to be useful, and we will likely need to transition to a relational domain, which is more likely to produce meaningful invariants for symbolic execution. Indeed, to efficiently cut paths, constraints passed to symbolic execution should closely mirror the conditions encountered in real programs. Since most conditions in real programs involve multiple variables, this results in relational constraints among these variables.

Trace partitioning Another technique of interest is trace partitioning, which involves refining the abstract state according to possible execution traces. When taken to its limits, this technique can transform abstract interpretation into a form of over-approximating symbolic execution, making it a natural area of study to bridge the gap between abstract interpretation and symbolic execution.

In our context, a typical use of this technique would be to prove that certain sets of traces are safe and do not need to be explored symbolically. However, it is important to find a trade-off to avoid duplicating the efforts of symbolic execution within the abstract interpreter. An open question regarding path partitioning is whether it can be used to approximate the number of loop iterations required to reach a bug.

Extensions This project opens the door to several extensions, two of which are described below:

Inference of preconditions As previously mentioned, once abstract interpretation detects a potential bug, we plan to use backward reasoning to infer numeric constraints, which will be passed to symbolic execution to reduce its search space. These numeric constraints can be viewed as unsafety preconditions, necessary to trigger the bug. However, backward reasoning may have limitations, particularly in the presence of loops. The inference of such preconditions is a broad area of exploration, and a potential starting point could be the use of fixpoint solvers employed in the SMT community.

Application to security analysis Several recent security analysis tools combine dataflow analysis, to identify potential flows from a compromised source to a vulnerable sink, with dynamic analysis, to validate whether a flow from that source to the sink actually exists, in order to detect security vulnerabilities in large software systems [7, 9]. Since abstract interpretation and symbolic execution respectively generalize dataflow analysis and dynamic analysis, it would be interesting to evaluate how our analyzer could be useful in vulnerability detection.

2 Motivating Examples

```
 1  int MIN_SIZE = 16;
 2
 3  void main() {
 4    int size = read_int();
 5    if (size < MIN_SIZE) {
 6      size = MIN_SIZE;
 7    }
 8
 9    int n = read_int();
10    if (n > size) {
11      n = size - 1;
12    }
13    int[] buffer = new int[size];
14    int i = 0;
15    while (i < n) {
16      i = i + 1;
17      buffer[i] = i;
18    }
19
20    do_something(buffer);
21    delete buffer;
22  }
```

Figure 1: Simple off-by-one error.

A simple example Figure 1 illustrates an example of an incorrect calculation that results in an off-by-one error. When the buffer size (line 4) and the number of integers to be written to the buffer (line 9) — both provided by the user — are equal and greater than the minimum buffer size, the loop initializing the buffer ends with an out-of-bounds write (line 17).

This example is interesting for several reasons, as it highlights the importance of relational invariants and the utility of trace partitioning. Regarding trace partitioning, an abstract interpreter could prove that if the condition at line 10 is satisfied, the rest of the program is safe, with no runtime errors. Consequently, the symbolic executor would only need to explore paths where this condition is not met.

For the relational invariants aspect, let us assume, for the sake of simplicity, that the buffer size is a constant equal to 16. A non-relational forward analysis would determine that at the beginning of the loop line 16, $n \in [0, 16]$ and $i \in [0, 15]$. When the write occurs at line 17, $i \in [1, 16]$ indicating a potential off-by-one error for $i = 16$. A backward analysis starting from line 17 with the constraint $i = 16$ would only infer that at line 16, $i \in [0, 16]$ meaning that $n \in [1, 16]$. While this constraint can be passed to the symbolic executor, it hardly prevents a single path from being explored.

In contrast, a relational forward analysis would easily infer that $0 \leq n \leq size$. Given that the invariant $0 \leq i < n$ holds at the beginning of the loop, we have $1 \leq i \leq n \leq size$ at line 17. Since writes are safe when $0 \leq i < size$, an off-by-one error can only occur when $i = n = size$. The constraint $n = size$ can be passed to symbolic execution, effectively narrowing the exploration to the single path that leads to the bug!

Note that while all state-of-the-art sound abstract interpreters detect this potential buffer overflow, none of them can produce a trace leading to the out-of-bounds write. Although state-of-the-art symbolic executors can find the bug, they require exploring a number of paths that is linearly proportional to the minimum buffer size.

```
1  struct CELL {
2    int content;
3  }
4
5  struct BUFFER {
6    CELL[] arr;
7    int len;
8    int pos;
9  }
10
11 BUFFER init(int length) {
12   BUFFER buf = new BUFFER;
13   buf.arr = new CELL[length];
14   buf.len = length;
15   buf.pos = 0;
16   return buf;
17 }
18
19 void push(BUFFER buf, int value) {
20   CELL cell = new CELL;
21   cell.content = value;
22   buf.arr[buf.pos] = cell;
23
24   if (buf.pos < buf.len - 1) {
25     buf.pos = buf.pos + 1;
26   } else {
27     buf.pos = 0;
28   }
29 }
30 int pop(BUFFER buf) {
31   if (buf.pos > 0) {
32     buf.pos = buf.pos - 1;
33   } else {
34     buf.pos = buf.len - 1;
35   }
36
37   int result = buf.arr[buf.pos].content;
38   delete buf.arr[buf.pos];
39   return result;
40 }
41
42 void main() {
43   BUFFER buf = init(16);
44   int cnt = 0;
45   int n;
46
47   while (cnt >= 0) {
48     n = read_int();
49     if (n > 0) {
50       push(buf, n);
51       cnt = cnt + 1;
52     } else {
53       if (cnt > 0) {
54         print_int(pop(buf));
55       }
56       cnt = cnt - 1;
57     }
58   }
59
60   delete buf;
61 }
```

Figure 2: Overflowing ring buffer.

A difficult example Figure 2 shows a simple implementation of a ring buffer in the `While++` language. The ring buffer data structure consists of an array that stores the elements, the length of that array, and the position of the next cell to be used for storing elements. This data structure is accompanied by three functions: an `init` function to initialize the ring buffer, a `push` function to insert elements, and a `pop` function to remove elements from the buffer.

The `main` function of the program repeatedly prompts the user for input and, based on the response, either pushes or pops integers to/from the ring buffer. A counter (line 44) is used to prevent popping from an empty buffer. However, if more elements than the buffer's capacity are pushed, the counter will exceed the buffer's length, allowing more elements to be popped than the number actually stored in the buffer. This leads to a null pointer dereference in the `pop` function (line 37). It is worth noting that the `push` function may also cause memory leaks (line 22), but since these do not result in runtime errors, they will not be addressed here.

Here again, a sound abstract interpreter will report that the ring buffer may contain null pointers and, therefore, that the `pop` function could potentially dereference a null pointer. However, none of the state-of-the-art abstract interpreters can produce a trace leading to such a null pointer dereference. Similarly, while symbolic execution has proven to be highly effective at finding bugs, especially when combined with fuzzing, neither state-of-the-art symbolic executors nor fuzzers are able to generate a faulty trace leading to this error.

The primary reason for this disappointing result is pretty simple: the unique shortest path to the crash requires 34 consecutive correct choices, one path out of a total of 17 179 869 184 possible paths of the same length. This is a textbook case of the path explosion problem, which turns exploration-based analysis into a wild goose chase without a good map to guide the way.

Could our analyzer help detect this bug? This example is actually extremely challenging, and it would be a significant achievement to develop an abstract interpretation that generates preconditions to assist symbolic execution in identifying the bug. Indeed, in this case the useful constraints not only relate variables to one another but also connect program paths to those variables. To effectively guide symbolic execution toward the bug, the abstract interpretation must: 1) infer that the program remains safe as long as the counter does not exceed the buffer length; 2) conclude that all faulty traces must begin with at least 17 consecutive pushes.

These challenges are tough, and we are not even sure where the starting line is — so, needless to say, there is plenty of work ahead!

References

[1] Cristian Cadar and Koushik Sen. *Symbolic Execution for Software Testing: Three Decades Later.* 2013.

[2] Antoine Miné. *Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions.* 2014

[3] Antoine Miné. *Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation.* 2017.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu and Irene Finocchi. *A Survey of Symbolic Execution Techniques.* 2018.

[5] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn and Jules Villard. *Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic.* 2020

[6] Patrick Cousot. *Principles of Abstract Interpretation.* 2021.

[7] Aurore Fass, Dolière Francis Somé, Michael Backes and Ben Stock. *DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale.* 2021.

[8] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer and Peter W. O'Hearn. *Finding Real Bugs in Big Programs With Incorrectness Logic.* 2022.

[9] Jayakrishna Vadayath and Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, Christophe Hauser and Yan Shoshitaishvili. *Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs.* 2022.