Formal Verification of Compiled Binaries

Jason An - jasan@ucla.edu
Alexander Zhang - alexyz@ucla.edu
Omar Elamri - omarelamri@ucla.edu

Abstract

Formal verification provides a high level of assurance but comes with many drawbacks such as tooling, complex semantics, and lots of manual work. We perform formal verification on compiled binaries instead—using angr and the z3 SMT solver. Our verifier aims to detect vulnerabilities, verify correctness, support multiple architectures, and provide cross-language support. Results are promising—we are able to verify programs that implement simple integer operations with branches on both x86 and ARM, verify programs that use external library calls to read from standard input and catch a simple buffer overflow. Other works exist; however, they are fairly limited in scope—either being limited to x86 and/or needing manual lifting in order to work.

Introduction

Formal verification provides a very high level of assurance, but it is not commonly used because it is difficult to implement. Formal verification has previously been performed on source code, but this has several drawbacks. First, formally verifying source code is inherently language-dependent. Tooling needs to be customized for each programming language, and verifying programs written in multiple languages such a C and Rust is difficult. Second, programming languages tend to have complex semantics. Manual work is needed to properly model the program, and mistakes can undermine the results of the verification. In our project, we attempt to perform formal verification on compiled binaries using the angr concolic execution engine. We first concolically execute the binary with angr, then run the z3 SMT solver on the symbolic results and user-inputted invariants to prove the program's correctness. We aim to create a verifier that can detect vulnerabilities and verify correctness with support for multiple architectures and programming languages.

Project Goals & Timelines

This project has four primary goals.

1. Detecting vulnerabilities such as buffer overflows: As buffer overflows are a fairly common vulnerability, being able to find these would reduce attack vectors substantially.
2. Verifying output correctness: A common way to "verify" correctness is to use unit tests that check for specific values. However, unit tests are only as expansive and thorough as their writer. Using a symbolic execution engine will find paths potentially not found by unit testing.
3. Support for multiple architectures: The aim is to provide a general verification system—not one tied specifically to x86, ARM, or RISC-V.
4. Cross-language support: By formally verifying the binary and not the source code, this project can verify programs language-agnostically—not needing to work with each language's intricacies (subject to ABI complexity).

As this project is fairly complicated, there are multiple goals that we set out to complete. To date, the following are completed:

- Verification of simple integer computations
- A simple-to-use but flexible API to interact with the verifier
- Verification of programs utilizing library calls and standard input
- Verification of return address preservation to catch buffer overflows
- Verification of multiple programming languages, including C and Rust
- Verification of multiple architectures, including x86 and ARM

In the future, we may look towards:

- Verification of floating-point computations
- Detection of out-of-bounds memory operations not just restricted to the saved return address of the function

Results

Integer Operations

We started by implementing a verifier for simple functions that take an integer argument and return an integer value computed from the argument. We used the following functions for testing:

```
int div2good(int x) { return x / 2; }
int div2good2(int x) { return (x + ((x >> 31) & 1)) >> 1; }
int div2bad(int x) { return x >> 1; }
int div2contrived(int x) {
    if (x == 0x13371337) {
        return x / 2 + 1;
    } else {
        return x / 2;
    }
}
```

These functions are all supposed to divide their argument by 2 and return the truncated quotient. The `div2good` function does so using the C division operator, while the `div2good2` function uses bitwise operations. The `div2bad` function uses a single-bit shift, which works for nonnegative numbers but returns the wrong value for negative numbers since it does not truncate properly. The `div2contrived` function divides using the C division operator like `div2good`, but if the argument is `0x13371337` it adds one to the resulting value.

We compiled the C code using GCC, then we ran our verifier on the functions with a constraint requiring the return value to be equal to the argument divided by 2. The verifier successfully verified `div2good` and `div2good2` and found counterexamples for `div2bad` and `div2contrived`:

```
Successfully verified rule __main__.rule_div2 for div2good
Successfully verified rule __main__.rule_div2 for div2good2
Failed verifying rule __main__.rule_div2 for div2bad (constraints
correct): arguments [-0x3918791d] return -0x1c8c3c8f
Failed verifying rule __main__.rule_div2 for div2contrived
(constraints correct): arguments [0x13371337] return 0x99b899c
```

Note that our verifier alerts the user of exactly which constraint was not satisfied.

We also tested our verifier on functions containing loops, which can't be easily verified with normal symbolic execution unlike the division functions:

```
int triangularNum(int n) {
    if (n > 100) {
        return -1;
    }
    int sum = 0;
    for (int i = 0; i < n; i ++) {
        sum += i;
    }
    return sum;
}

int triangularNumBad(int n) {
    if (n > 100) {
        return -1;
    }
    char sum = 0;
    for (int i = 0; i < n; i ++) {
        sum += i;
    }
    return sum;
}
```

These functions compute triangular numbers using a simple `for` loop. The `triangularNumBad` function uses a `char` instead of an `int` to store the sum, resulting in overflows for larger values of `n`. The verifier successfully verifies `triangularNum` and finds many counterexamples for `triangularNumBad`:

```
Successfully verified rule __main__.rule_trinum for triangularNum
Failed verifying rule __main__.rule_trinum for triangularNumBad
(constraints correct): arguments [0x11] return -0x78
Failed verifying rule __main__.rule_trinum for triangularNumBad
(constraints correct): arguments [0x12] return -0x67
Failed verifying rule __main__.rule_trinum for triangularNumBad
(constraints correct): arguments [0x13] return -0x55
...
```

These results show that our verifier is capable of verifying code containing arithmetic and logical operations as well as control constructs like `if` statements and loops.

Standard Input, Library Calls, and Buffer Overflows

Our verifier is also able to interface with standard input and work with library calls. Here's a toy example that uses libc's `read` function:

```
int testStdin() {

   signed char num;

   read(0, &num, 1);
   return num / 2;
}
int testStdinBad() {
   signed char num;
   read(0, &num, 1);
   return (num / 2) + 1;
}
```

The first function reads in a byte from standard in and places it in the num buffer. It returns the byte divided by two—similar to div2good but retrieves its input elsewhere. The second function performs the same operation but adds + 1 to make it incorrect. Our verifier should verify that the first function does divide by two, and the second one should fail:

```
Successfully verified rule rule_stdin for testStdin

Failed verifying rule rule_stdin for testStdinBad (constraints by2):
arguments [] stdin 0xff000[...] return 0x1
```

With the addition of standard in as a constraint, our verifier is much more flexible. We'll use this functionality to detect a buffer overflow originating from an extended read. To detect a buffer overflow, our verifier stores the return address before emulating execution. After execution, it compares the current return address and fails to verify if they differ.

```
int testStdinBof() {
   signed char num;
   read(0, &num, 64);
   return num / 2;
}
```

This function works similarly to the other two, but now reads in a maximum of 64 bytes. Since our num buffer is only 1 byte long, our verifier should fail on a longer input:

```
Failed verifying rule rule_stdin for testStdinBof (constraints
preserves return address): arguments [] stdin
0x01000000000000000038007f0[...] return 0x0
```

When passing in an especially large value to standard input, the return address is overwritten, and our verifier catches it.

Cross-language, cross-architecture support

Our verifier works across languages and architectures. In the next example, we use an executable that interfaces with a Rust library—all compiled for `aarch64`.

Take this toy example of a Rust library:

```
use std::ffi::c_int;

#[no_mangle]

pub extern "C" fn is_negative(x: c_int) -> c_int { (x >> 31) & 1 }

#[no_mangle]

pub extern "C" fn is_negative_bad(x: c_int) -> c_int { (x >> 31) & 2 }
```

We have two functions that detect if a number is negative—the second one fails since it doesn't check the signed bit. Now, take an executable that interfaces with these Rust functions:

```
int is_negative(int x);

int is_negative_bad(int x);

int div2rustgood(int x) { return (x + is_negative(x)) >> 1; }

int div2rustbad(int x) { return (x + is_negative_bad(x)) >> 1; }
```

Our verifier should verify the third function and fail on the fourth function since it uses the correct and incorrect `is_negative` functions respectively.

```
Successfully verified rule aarch64_rule_div2 for div2rustgood

Failed verifying rule aarch64_rule_div2 for div2rustbad (constraints
correct): arguments [-0x4745a800] return -0x23a2d3ff
```

Our verifier caught the issue—in a language other than C and an architecture other than x86.

Limitations

Our verifier works on a wide variety of software features; however, there are clear limitations. For example, it doesn't work on more complicated library functions—functions that used libc's `fgets` failed to verify no matter the function body. In addition, while our verifier mostly works cross-architecture, return address semantics differ for each architecture. Hence, those are cases we'd need to handle manually.

Other limitations have to do with our toolchain. `angr`, which uses concolic execution, is susceptible to state explosion if branching is especially complex in a function. `angr` and `z3` are not fully modeled theorem provers, unlike ACL2 and coq. However, our solution is simple and extensible to many scenarios—providing an API that specializes in flexibility and that is "turn-key" to fit many needs.

Related Works

Many works exist that attempt to verify compiled binaries formally. However, they're particularly limited in scope.

In Bockenek et. al.'s [1] paper on Formal Verification of Memory Preservation of x86-64 Binaries, they attempt to verify language agnostically (like this project) by verifying the compiled binaries. As a limiting factor, they only focus on memory preservation instead of general verification. This requires proving "the absence of common memory-related issues, such as buffer overflows or some forms of data leakage" (Bockenek, 2019). They do this by studying the x86-64 ISA and developing invariants that memory accesses/writes may not violate. While this has the advantage of potentially preventing buffer overflows, this approach is inherently tied to x86-64—where architecture invariance is one of our goals. It also checks only a small subset of program correctness relating to memory correctness, without verifying that the program behaves correctly too.

In Goel et. al.'s [2] paper on the simulation of x86 programs that make system calls, they approach the issue of nondeterminism by separating their analysis into logical and execution modes. In the logical mode, all x86 logic is pure where any external state is modeled as a changing environment variable. The major drawback of this solution is that it's limited to x86 (as is the previous solution) and that it needs to "lift" the program into another HOL solver (Goel, 2014).

In Goel's [3] later thesis defense, they discuss a newer verifier they've been working on which models the entire x86 pipeline including instruction fetching, decoding, and executing in ACL2. This is much more powerful than the previous HOL solver, and also no longer needs an external lifting script to function properly.

Other methods of verification don’t meet this project’s needs. This project’s verification method is fairly simple and supports a wide variety of applications and environments. It may not be as encompassing or powerful as the x86-specific solver, but it's much easier to work with and use with other operating systems or architectures.

References

[1] JA. Bockenek, F. Verbeek, P. Lammich, and B. Ravindran. “Formal Verification of Memory Preservation of x86-64 Binaries,” 2019 Computer Safety, Reliability, and Security: 38th International Conference, SAFECOMP 2019, Turku, Finland, September 11–13, 2019, Proceedings. Springer-Verlag, Berlin, Heidelberg, pp. 35–49, doi: https://doi.org/10.1007/978-3-030-26601-1_3

[2] S. Goel, W. A. Hunt, M. Kaufmann and S. Ghosh, "Simulation and formal verification of x86 machine-code programs that make system calls," 2014 Formal Methods in Computer-Aided Design (FMCAD), Lausanne, Switzerland, 2014, pp. 91-98, doi: https://doi.org/10.1109/FMCAD.2014.6987600

[3] Goel, S. (2016). Computer Architecture and Program Analysis Formal Verification of x86 Machine-Code Programs. https://www.cs.utexas.edu/~byoung/cs429/shilpi-slides.pdf