Static Sign Analysis Using Angr

Hao Ke
New York University

Munieshwar (Kevin) Ramdass
New York University

Anthony Masi
New York University

Abstract

Static analysis is a subset of binary analysis, allowing the examination of a program's various branches of execution without actually running it. Static sign analysis, one possible methodology implemented in binary analysis, seeks to determine whether the contents of a register or a stack variable are positive, negative or unknown for all the possible branches of execution available. This approach can be used to predetermine program behavior and uncover possible bugs. The problem of static sign analysis is still a work-in-progress, so we've proposed a solution which detects possible divide-by-zero bugs in applications. Our work is based upon the well-known open source static analysis framework, angr. We achieved the goal of static sign analysis by generating and traversing a control-flow graph for our input application, keeping track of possible integer signs from the root node of the graph through its successors. We demonstrate the validity of this approach through a proof-of-concept display on a binary with a possible divide-by-zero error present. Our technique allows us to rapidly recognize and identify these types of errors in applications.

1 Introduction

Sign analysis is the programmatic determination of integer signs for variables and expressions in a binary. A sign can be positive, negative, zero or unknown. A sign is deemed unknown if we cannot evaluate it as a variable or expression. This may occur if a register is not used or a stack variable has been initialized without a value. Sign analysis aids in vulnerability discovery by providing a detection mechanism for arithmetic-based logic bugs, such as divide-by-zero errors.

One of angr's developers, Andrew Dutcher[1], successfully performed sign analysis on a small program consisting of basic conditional statements using symbolic execution. The control flow graph (CFG) generated was used to obtain basic blocks representing all possible paths through the program's execution. An abstract syntax tree (AST) was used to trace the binary's operations. Symbolic variables were then created and evaluated as per operations in the AST. The signs of the resulting values were outputted allowing his solution to catch variables that both evaluated to zero and were used in division operations.

For our project, we chose to use the static analysis method to approach this problem. We chose not to execute the binary but to base our algorithm upon information gathered while traversing the program's control flow graph. Instead of trying to obtain the exact, concrete value of each program state, we used the abstraction of signs to determine possible signs of a register or stack variable at any given program state. Otherwise we combined different possible sign result based on lattice theory.

2 Related Work

Although intraprocedural analysis has been proven to be both a computationally hard and NP-complete problem, work on designing more efficient static analysis tools has continued[1]. Landi's designation that approximation was necessary to construct reasonably good analyses is the reasoning behind a multitude of static analysis tools lacking in comprehensiveness and code coverage area[2]. Other solutions have been developed, which have either focused on simplistic bugs in a single language[5], a specific type of malware[4], or a specific algorithm with known flaws in terms of code coverage[6]. All of these individual solutions are valuable contributions as proof-of-concept but are lacking in the comprehensiveness necessary to be adopted as day-to-day utilities

In response to this, we have elected to build upon the binary analysis framework angr due to its versatility in technique for identifying vulnerabilities in binaries[7]. Angr's ability to generate control flow graphs as a collection of objects has made it an ideal candidate for this type of work. Previous research done on sign analysis has also proved useful, although much of it focused mainly upon analyzing source code rather than binaries[8]. Other previous research on sign analysis[9] and lattice theory[10]may be harnessed in conjunction with Kuru and Gungor's work to analyze binaries for vulnerabilities with the increased code coverage provided by angr.

We have found inspiration from other work based upon the abstraction and detection of vulnerabilities in binaries. Recovery of variable types has been thoroughly covered by Lee, et al.[11]. Recovery of intermediate representations is a crucial component of our proposed solution. Similarly, source code based analysis of possible software vulnerabilities has been researched by Cadar, et al[12] and serves as a building block for our analysis algorithm. Static analysis of binaries for software vulnerabilities has been touched upon by Molnar, et al.[13] as well as a whitepaper written by security firm Grammatech describing a closed source framework they have designed[14]. Although inspiration was drawn from both Molnar, et al and the Grammatech paper, we chose to go about our research in a different way.

Bowdidge, et al's analysis of why static analysis tools are generally avoided by developers inspired our work as well, as he presents evidence that many developers prefer to avoid these tools due to the fact that they don't assist in the actual patching of bugs[15]. We hope to use this as inspiration to produce a plug-in that not only identifies vulnerabilities in binaries but also determines where exactly the vulnerability has occurred.

3 Background

In binary analysis, we used control flow graphs to represent the execution flow of a given program. Control flow graphs can also describe the relationship between blocks of code in a program. Each node in the control flow graph is defined as a basic block. The key features of a basic block are (1) there is no jump or jump target within each basic block and (2) each basic block has exactly one entry and exit.

In order to make it easier for us to analyze the binary, we first compute an intermediate representation. An intermediate representation is the code used internally by a compiler or virtual machine to represent source code. This intermediate representation can be parsed to detect the presence of potential undefined behavior, such as divide-by-zero errors.

4 Design

As stated in Section 1, we chose to implement static analysis in our solution rather than symbolic execution. We had to analyze the values of the registers presented to us in angr's intermediate representation, as well as the values at the memory addresses found in each control flow graph node. Analyzing the values of these registers and stack variables allows us to detect which execution paths may lead to a divide-by-zero error.

Since we targeted this type of error specifically, we can simply flag registers or stack variables with the value of 0 that are later used as the divisor of a division operation. Since we can obtain the intermediate representation (IR) of each basic block, we can access register and stack variable values, as well as instructions executed on said values. Our analysis is reliant upon parsing the intermediate representation of a binary in order to keep track of variable sign information.

5 Implementation

Our choice of implementing static analysis techniques over symbolic execution results in an inability to obtain exact values for our various states of program execution. Therefore, we chose to emulate the stack using an assumed address, allowing our algorithm to handle memory operations such as LOAD or STORE. Using the mem, mem_const, and reg_constant functions of angr, we were able to keep track of constant values in an effort to accurately track integer sign.

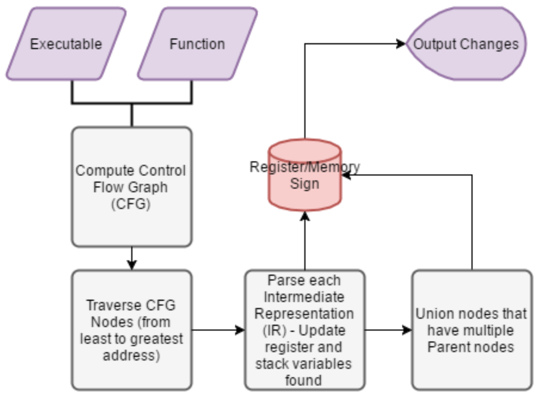

Executable
Function
Output Changes
Compute Control Flow Graph (CFG)
Register/Memory Sign
Traverse CFG Nodes (from least to greatest address)
Parse each Intermediate Representation (IR) - Update register and stack variables found
Union nodes that have multiple Parent nodes

Figure 1: Our algorithm's output.

To calculate whether a divide-by-zero error exists, we begin by computing the control flow graph for the target function. Our program than traverses each node of the control flow graph iteratively, parsing each intermediate representation statement generated by angr and updating status accordingly. A "0" indicates a value of 0, a "1" indicates a positive value, and a "2" indicates a negative value. The empty set, denoted as "[]" is used to indicate an unknown variable or a variable that has not yet been analyzed. If an IR instruction is a WrTmp instruction, we obtain the binary operation (BinOp) sub-instruction which uses two variables in an operation. We then evaluate this to determine the sign of the result or the division-by-zero error. An IR instruction can also be a PUT or STORE instruction in which we would assign a value to a register or update a register value. If the current node has multiple predecessor nodes, we perform analysis based on a unioned result set. After determining a value's sign, we output this determination to the user. Below is the pseudocode for our algorithm which handles the WrTmp instruction:

Algorithm 1 Sign Analysis

```
1: procedure $sign\_analysis(CFG)$
2:     $registers \leftarrow$ get register values of $IR$
3:     $memory \leftarrow$ get stack values of $IR$
4:     $CFG \leftarrow$ sort the $CFG$
                                        ▷ Loop CFG nodes
5: loop:
6:     if $CFG(i)(PREDECESSORS) > 1$ then
7:         $UNION\_PREDECESSORS$
8:     $IR = get\_IR(CFG[i])$
                                        ▷ Loop IR Instructions
9: loop:
10:    if $instruction[i] == DivModS64to32$ then
11:        $output$ "Divisionby0"
12:    $update\_register\_and\_memory(memory, instr)$
```

6 Evaluation

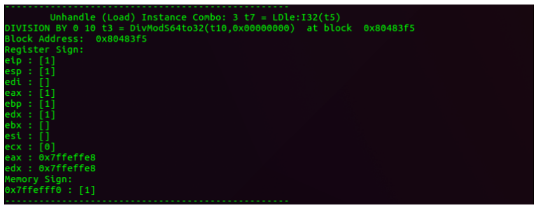
```
//test.c

int main(int argc, char ** argv) {
int a, b, c;
a = -42;
b = 0;

if (argc < 1) {
    c = a / b;
} else {
    c = a + b;
    }

    return c;
}
```

From test.c, we have assigned variable ‘b’ to 0. The first if-condition assigns the result of a division operation to variable ‘c’. Variable ‘b’ is the divisor and we expect our analysis to output that a division-by-zero operation is occurring. To compile using GCC, we have added the ‘-m32’ option to handle only 32-bit instructions.



```
--------------------------------------------------
       Unhandle (Load) Instance Combo: 3 t7 = LDle:I32(t5)
DIVISION BY 0 10 t3 = DivModS64to32(t10,0x00000000)  at block  0x80483f5
Block Address:  0x80483f5
Register Sign:
eip : [1]
esp : [1]
edi : []
eax : [1]
ebp : [1]
edx : [1]
ebx : []
esi : []
ecx : [0]
eax : 0x7ffeffe8
edx : 0x7ffeffe8
Memory Sign:
0x7ffefff0 : [1]
--------------------------------------------------
```

Figure 2: Our algorithm’s flowchart.

Our result, as shown in Figure 2, displays information from the basic block where the division-by-zero bug is located. Most of the registers have been updated with a sign representation. According to our calculations, the variable ”b” in test.c was loaded into register ”ecx”, which was used in the ”DivModS64to32” operation as the divisor. Our analysis looks for divisors that are equal to 0 and outputs a division-by-zero alert, so naturally this sequence was flagged correctly.

7 Limitations

Our project is limited solely by its ability to successfully compute sign information for a single target function. Additionally, we have chosen to only target 32 bit architectures for the time being, so we are currently limited by the availability of parsers for new instruction sets.

8 Future Work

As mentioned in Section 7, our script can currently only analyze specific functions. Information would be omitted if multiple inline callee functions within the initial target function are present. Future work may include extending the scope of the sign analysis, extending it to handle multiple callee functions as well. Due to our time limit, we implemented this sign analysis program with only 32-bit compatibility. We could further extend our work by making it compatible with various other architectures.

9 Conclusion

Using static sign analysis, it is possible to determine whether the contents of a register or a stack variable are positive, negative, or unknown for all the possible branches of program execution. We used this approach to implement a vulnerability discovery script, capable of detecting potential undefined behavior in a binary in the form of divide-by-zero errors. Building upon the angr framework, we traversed the control-flow graph for our input application, tracking the sign of register and stack variable values. This proof-of-concept can be viewed on GitHub at the repository listed in Section 11.

10 Acknowledgments

We would like to thank Professor Brendan Dolan-Gavitt of New York University’s Tandon School of Engineering for the great semester and the authors of angr for making any of this possible.

11 Availability

Our algorithm is available on GitHub, located at:

`https://github.com/mramdass/Static_Sign_Analysis/`

References

[1] M. Might, “What is static program analysis.” `http://matt.might.net/articles/intro-static-analysis/`.

[2] W. Landi, “Undecidability of static analysis,” *ACM Letters on Programming Languages and*

Systems (LOPLAS), vol. 1, no. 4, pp. 323–337, 1992.

[3] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, vol. 68, pp. 18–33, 2015.

[4] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," tech. rep., DTIC Document, 2006.

[5] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[6] M. R. Parvez, "Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries," 2016.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[8] S. Kuru and T. Gungor, "Sign analysis technique for predicting system behavior," *International journal of intelligent systems*, vol. 7, no. 5, pp. 419–444, 1992.

[9] M. I. S. Anders Moller, "Static program analysis." `https://cs.au.dk/~amoeller/spa/spa.pdf/`.

[10] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[11] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[12] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, pp. 209–224, IEEE, 2008.

[13] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs.," in *USENIX Security Symposium*, vol. 9, pp. 323–351, 2009.

[14] "Eliminating vulnerabilities in third-party code with binary analysis," GrammaTech.

[15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 672–681, IEEE, 2013.

Notes

[1]Presentation available at http://panda.moyix.net/ moyix/v/AndrewDutcher-Angr.mp4