Report Number:

2019-12-04

Static Taint Analysis of Binary Executables Using Architecture-Neutral Intermediate Representation

Elaine Cole

Ghidra, National Security Agency's powerful reverse engineering framework, was recently released open-source in April 2019 and is capable of lifting instructions from a wide variety of processor architectures into its own register transfer language called p-code. In this project, we present a new tool which leverages Ghidra's specific architecture-neutral intermediate representation to construct a control flow graph modeling all program executions of a given binary and apply static taint analysis. This technique is capable of identifying the information flow of malicious input from untrusted sources that may interact with key sinks or parts of the system without needing access... **Read complete abstract on page 2.**

Static Taint Analysis of Binary Executables Using Architecture-Neutral Intermediate Representation

Elaine Cole

Complete Abstract:

Ghidra, National Security Agency's powerful reverse engineering framework, was recently released open-source in April 2019 and is capable of lifting instructions from a wide variety of processor architectures into its own register transfer language called p-code. In this project, we present a new tool which leverages Ghidra's specific architecture-neutral intermediate representation to construct a control flow graph modeling all program executions of a given binary and apply static taint analysis. This technique is capable of identifying the information flow of malicious input from untrusted sources that may interact with key sinks or parts of the system without needing access to the source code itself and can be retargetable to analyze the behavior of a given program across many different processors.

Static Taint Analysis of Binary Executables Using Architecture-Neutral Intermediate Representation

Elaine Cole
McKelvey School of Engineering
Washington University in St. Louis
elainemcole@wustl.edu

***Abstract*—Ghidra, National Security Agency's powerful reverse engineering framework, was recently released open-source in April 2019 and is capable of lifting instructions from a wide variety of processor architectures into its own register transfer language called p-code. In this project, we present a new tool which leverages Ghidra's specific architecture-neutral intermediate representation to construct a control flow graph modeling all program executions of a given binary and apply static taint analysis. This technique is capable of identifying the information flow of malicious input from untrusted sources that may interact with key sinks or parts of the system without needing access to the source code itself and can be retargetable to analyze the behavior of a given program across many different processors.**

***Index Terms*—control flow, , intermediate representation, program analysis, security**

I. Introduction

Whether someone is seeking to defend their own program or attack another's target program, program analysis in security seeks to identify vulnerabilities. There exist a wide variety of forms of program analysis, and we can categorize many into either static analysis, dynamic analysis, or a hybrid of the two. Static analysis itself focuses on the program as is, while dynamic analysis is done during the runtime of the program. A common issue that arises is a limitation of information regarding a target program: how might one analyze a program if they do not have all of the information that they might need or want in order to do certain types of analysis? Specifically, this project utilizes reverse engineering to execute analysis on a target program given only an executable, rather than the complete source code.

Software reverse engineering is a process of recovering key information of a program by working from the bottom up. Through this, we can apply program analysis. This allows threat intelligence researchers among others to work backwards on malware that they encounter in order to discover how and what the program is capable of attacking and even where it came from. There are various components of reverse engineering, and we leverage a decompiler to retrieve our desired program information.

A decompiler works in reverse as a compiler, taking in an executable file as input and attempting to create a high-level source file which can be recompiled successfully. Oftentimes, the re-created source file is not the same as the original program file which was used to compile the input executable; regardless, the decompiler extracts key program processes into human-readable structures.

We leverage Ghidra, a leading reverse engineering framework, and its provided decompiler to lift given executable files into higher-level intermediate representations (IRs) and then apply static analysis onto these IRs to identify vulnerabilities.

II. Ghidra

Ghidra [1] was developed by National Security Agency (NSA) and marks the first time that a tool of its caliber will be available for free to the public. Its existence was first revealed to the public through in March 2017 [2] and the source code was published to GitHub on April 4 2019 [3].

Ghidra is a customizable environment, that supports multiple platforms and instruction sets and is capable of disassembly, assembly, decompilation, graphing and scripting, and many other features. It uses its own IR language known as p-code [4] which is machine independent and designed to model general purpose processors. We can apply analysis to this middle ground between low-level machine-specific assembly and high-level programming languages in order to discover meaningful information and vulnerabilities of a given program without having been provided access to the original source code.

Within p-code representation, all data is manipulated explicitly, meaning that instructions have no indirect effects. This is vital for ease of use of p-code, as it avoids excess complexities of additional data manipulation that might not be considered intentionally. We use this representation of the target program to recover its control flow graph (CFG), a requirement of many forms of static analysis.

Our intent with this project was to recover the control flow and provide an extendible structure for future work of analysis to be built upon it. Much of the time spent required development of our knowledge of p-code itself, as there are multiple categories of operations. The list of categories is as follows:

- Data Moving
- Arithmetic
- Logical
- Integer Comparison
- Boolean
- Floating Point
- Floating Point Compare

- Floating Point Conversion
- Branching
- Extension and Truncation
- Managed Code

One of the requirements were to determine which categories were of use in the reconstruction of the CFG, which we discuss in the following section.

III. Control Flow

A control flow graph (CFG) is a directed graph which represents the computation and flow of control within a given program. Within it, a node represents a basic block, a segment of consecutive statements that has no branching in or out except at its beginning and end. This handles the computations. Edges represent possible flow of control between nodes. There may be more than one incoming or outgoing edge for each node.

Specific to p-code, a branch operation is a control flow out of a node, and a label is a flow into a node. At a high level, a node begins at a label instruction or after a jump instruction. Likewise, a node ends at a jump instruction or before a label instruction.

In the example below shown in "Fig. 1", we offer a simple C program with a conditional branch.

```
x = 6;
y = x + 4;
if (a) {
    z = y + x;
} else {
    x = x + 1;
}
z = x;
```

Fig. 1. Example C program with conditional branch.

As shown, the condition `a` determines the subsequent statement to be executed. The created CFG for this C program example is shown in "Fig. 2".

Here, the validity of the condition `a` decides which node to continue to. Where the flow of control travels to (or doesn't travel to) will affect the value of variable `z` in this example program. This is only a small demonstration of the importance of control flow.

IV. Implementation

Ghidra provides many useful tools within its own UI. However, Ghidra can also be run in headless mode from the command line, which allows for batch processing and is more extensible for larger target scopes. While Ghidra provides a

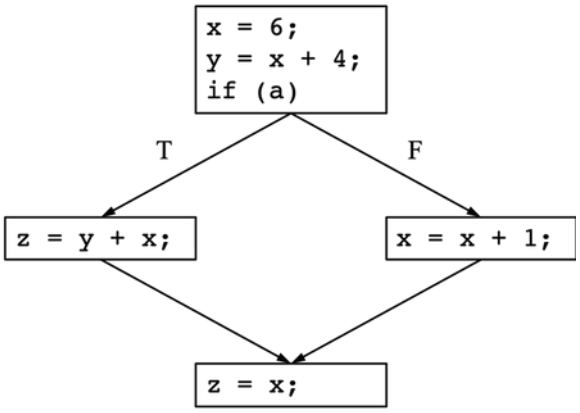

Fig. 2. CFG for Example C program.

graph view of control flow, we instead choose to develop our own structure to track the flow of control for easier use using Ghidra as a headless analyzer. We build our project as a Java script, which can be run in both the UI and via the command line with Ghidra.

Specifically, we develop an adjacency list structure for the CFG. In this, we design a data structure to hold nodes, which holds HashSets of both incoming and outgoing flows. Additionally, it holds a `CodeBlock` structure, the sequence of statements within the specific node. The edges are referred to as flows and have structures which hold both a `startNode` and an `endNode`, as each edge may only have one start and one end.

The process of development included two major steps: first, we partition the code into basic blocs of p-code instructions. Then, we mark the branches within the p-code as flows of control. From here, we develop a structure for pulling the control flow of a given executable file.

V. Taint Analysis Proof-of-Concept

Taint analysis is the analysis process of tracking information flow between untrusted (tainted) sources and trusted (untainted) sinks. These sources may be user inputs or network incoming sources in which we cannot trust that these are secure. Sinks, on the other hand, may refer to function calls to malloc in which malicious or tainted data may cause harm to the program or the environment. Taint analysis may be done dynamically, but for the purposes of proof-of-concept in this project, we implement static taint analysis.

We first define a taint policy. A taint policy defines how new taint is introduced, how taint propagates as instructions execute, and how taint is checked. We also define the sources and sinks, as well as the granularity of our taint analysis: is this byte-level or bit-level?

Using Ghidra's scripting tool, we build a script in Java that takes in two .txt files which list, line by line, the sources and sinks, respectively. The script marks our specified sources as tainted and then uses our control flow structure to track and propagate taint through the nodes. In this sense, our script

is extendible for different categories of sources and sinks. We also include a command line option to turn off script messages for usability.

VI. Usage

Our code repository can be found at https://gitlab.com/shellb34r/ghidraaa [5]. Ghidra's headless analyzer can be used through the command line using the format found in "Fig. 3" with further details found online in their readme [6].

```
<project location> <existing project name> -scriptPath <script path>
    -postScript ControlFlow.java -process <test file name>
```

Fig. 3. Example usage of Ghidra's Headless Analyzer.

Additionally, our control flow Java script can be imported as an extension into other scripts [7]. Our hope is that future work will be implemented and built upon this tool.

VII. Learnings and Future Work

Within our proof-of-concept taint analysis, we did not apply any taint sanitization in which we would remove taint. Because of this, our taint policy overtaints, marking untainted safe values as tainted. Future work would involve extending our taint policy to include additional sources and sinks as well as a more fine-grained taint level.

In addition, we hope that our control flow adjacency list can be extended upon in future work to include forms of program analysis beyond taint analysis. These may include symbolic execution and fuzzing integration.

VIII. Conclusion

Using Ghidra, we demonstrated the viability and value of extracting the control flow through Ghidra's IR language, p-code, to apply static analysis to a given target program. Ghidra is still a new tool, having been publicly released only this year. The deep and extensive object-oriented API provide a strong and powerful framework for reverse engineering, but a steep learning curve during the first months since its release.

We successfully build an extensible program that handles the groundwork control flow for further program analysis, and it is our hope that others will use this to further develop within the realm of IR-based vulnerability evaluation.

Acknowledgment

I send my special thanks to Professor Ning Zhang for his guidance.

References

[1] Ghidra, https://ghidra-sre.org/.
[2] Website, "Wikileaks," https://wikileaks.org/ciav7p1/cms/page_51183656.html, online.
[3] Website, "Ghidra," https://github.com/NationalSecurityAgency/ghidra, online.
[4] Website, "P-Code Reference Manual," https://ghidra.re/courses/languages/html/pcoderef.html, online.
[5] Website, https://gitlab.com/shellb34r/ghidraaa/tree/control-flow/, online.
[6] Website, "Headless Analyzer README," https://ghidra.re/ghidra_docs/analyzeHeadlessREADME.html, online.
[7] Website https://github.com/NationalSecurityAgency/ghidra/issues/243, online.