CSAW ESC Team B Spring 2021 Report

Spencer Hua (Student)
Vertically Integrated Projects
Georgia Institute of Technology
spencerhua@gatech.edu

Taleb Hirani (Student)
Vertically Integrated Projects
Georgia Institute of Technology
thirani7@gatech.edu

Siddhant Singh (Student)
Vertically Integrated Projects
Georgia Institute of Technology
ssingh484@gatech.edu

John Moxley (Student)
Vertically Integrated Projects
Georgia Institute of Technology
jmoxley@gatech.edu

Allen Stewart (Advisor)
Vertically Integrated Projects
Georgia Institute of Technology
allen.stewart@gtri.gatech.edu

***Abstract*—This report details the efforts of the Georgia Tech's Embedded System Cybersecurity VIP team to create a Ghidra script for use in the static analysis of binaries for various architecture. This tool uses Ghidra's plugin system and external engines to symbolically solve for constraints and reach a solution state in the loaded binary. The team demonstrates this tool and the process of creating and using it.**

I. Introduction

Throughout this paper, the Georgia Tech Embedded System Cybersecurity VIP team details the motivations and specifics in creating a Ghidra [1] script which can be leveraged to more effectively analyze binaries. There are several different engines which are used to drive the tool. Each of these can be used to perform a different analysis task. For example, one engine leverages the use of Sice Squad [2]'s RISC-V symbolic execution project to allow a researcher to perform symbolic analysis on RISC-V binaries within Ghidra. Other engines are incorporated as well, such as the Z3 Theorem Prover [3]. Furthermore, the usage of this tool is intended to be intuitive through the use of Ghidra's GUI elements. Ultimately, this tool makes binary analysis a much easier research task.

II. Inspiration

The inspiration to develop a Ghidra [1] script, as well as write documentation to help make Ghidra plugin development more accessible, came from two open source projects. Ghidra-Emu-Fun is an open source Ghidra P-code emulation front-end project developed by the students in the TRX CTF team of the Sapienza University of Rome. This project is a Ghidra plugin, developed in Python, that serves as an accessorized front-end to the P-code emulation feature provided by the Ghidra RE framework for the ARM architecture. This front-end supports common debugging commands as well as enabling support for function hooking, wildcard bytes for function parameters as well as automated fuzzing via batch commands. By reviewing the code written by the TRX team, the team was able to further its understanding of Ghidra P-code lifting and associated plugin development practices. Another tool that served as inspiration and as a base for the team's tool development was the RISC-V Symbolic Execution engine developed by the Sice Squad CTF team. This tool was developed as a custom tool to solve the CSAW ESC 2020 challenge. It consists of a simply disassembly and lifting script combined with a small symbolic execution engine, and was put together to aid in reverse engineering RISC-V 32-bit binaries. The engine is able to solve for basic symbolic constaints with the help of the Z3 Theorem Prover, and was instrumental in being succinct enough to be easily slotted into different frameworks.

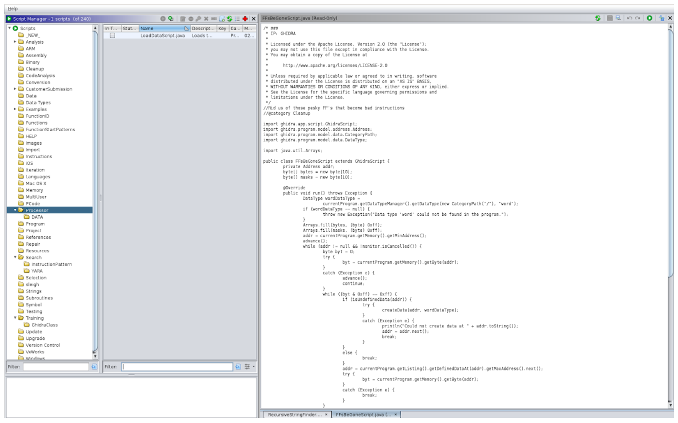
III. Ghidra Scripts vs. Extensions

In order to effectively extend Ghidra's functionalities, it is important to understand the proper way to do so. Ghidra offers a variety of interfaces in which to write extensions. Being an open source project [1], it is possible to change any aspect of Ghidra, even the core functionality. However, the Scripting Manager and Extensions provide far more convenient ways to modify small aspects of Ghidra's behavior without needing to delve into the core code.

Ghidra scripts provide this easy way to modify Ghidra's behavior. They can be written in either Python or Java, and the Script Manager includes a built-in GUI editor to manipulate and run them, as well as a variety of example scripts in both languages, which serve as good starting points for new script developers. As seen in Figure 1, scripts in the Script Manager are organized into categories on the left, while the editor can be seen on the right.

Ghidra extensions (also known as plugins or modules), on the other hand, are Java programs that are intended for more heavy-weight functionality and changes to Ghidra. Extensions are run alongside Ghidra's main instance, and can heavily modify Ghidra itself, not being limited to working with Ghidra's current program. It is recommended that Ghidra extensions be developed using Eclipse [4] with the recommended development packages installed for easier use [5].

It is important to note that Ghidra's full program API is accessible for scripting programs. Thus, the team chose to implement the tool as a Ghidra script written in Python, as there are no limitations with this method. Furthermore, several projects that the team intends on interfacing with, such as the RISC-V Symbolic Execution Engine, already exist as Python


Script Manager
FFsBeGoneScript.java (Read-Only)
RecursiveStringFinder....
FFsBeGoneScript.java (...

```java
/* ###
 * IP: GHIDRA
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
//Rid us of those pesky FF's that become bad instructions
//@category Cleanup

import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.data.CategoryPath;
import ghidra.program.model.data.DataType;

import java.util.Arrays;

public class FFsBeGoneScript extends GhidraScript {
        private Address addr;
        byte[] bytes = new byte[10];
        byte[] masks = new byte[10];

        @Override
        public void run() throws Exception {
                DataType wordDataType =
                        currentProgram.getDataTypeManager().getDataType(new CategoryPath("/"), "word");
                if (wordDataType == null) {
                        throw new Exception("Data type 'word' could not be found in the program.");
                }
                Arrays.fill(bytes, (byte) 0xff);
                Arrays.fill(masks, (byte) 0xff);
                addr = currentProgram.getMemory().getMinAddress();
                advance();
                while (addr != null && !monitor.isCancelled()) {
                        byte byt = 0;
                        try {
                                byt = currentProgram.getMemory().getByte(addr);
                        }
                        catch (Exception e) {
                                advance();
                                continue;
                        }
                        while ((byt & 0xff) == 0xff) {
                                if (isUndefinedData(addr)) {
                                        try {
                                                createData(addr, wordDataType);
                                        }
                                        catch (Exception e) {
                                                println("Could not create data at " + addr.toString());
                                                addr = addr.next();
                                                break;
                                        }
                                }
                                else {
                                        break;
                                }
                                addr = currentProgram.getListing().getDefinedDataAt(addr).getMaxAddress().next();
                                try {
                                        byt = currentProgram.getMemory().getByte(addr);
                                }
                                catch (Exception e) {
                                        break;
                                }
                        }
```

Fig. 1. The Ghidra Script Manager

projects. Thus, creating a Python script simplifies both the implementation of the team's code and the interfacing with other engines, making for a smooth and fast development experience.

IV. Ghidra-Bridge

One issue the team encountered is that Ghidra provides Python support through an internal Jython interpreter, which has no access to external site packages provided by Python. After some experimentation, the team discovered a solution in the form of Ghidra-Bridge [6].

Ghidra-Bridge is a free and open source tool based on the JFX-Bridge project, allowing a user to access the Ghidra APIs from external Python installations via Remote Procedure Calls (RPC). Thus, a Python3 script can be created with all required modules pre-installed, which can simultaneously access the Ghidra API through Remote Procdure Calls to the Ghidra-Bridge instance running on Ghidra's Jython interpreter. Using Ghidra-Bridge also allows the team to make use of a virtual environment, which is a boon for managing Python projects that require multiple, different dependencies, some of which may conflict with other projects. For the tool,the team used a powerful internal feature of Ghidra-Bridge's RPC server to enable a script run from within Ghidra to spawn an external Python script, creating a usage experience that is perfectly seamless to the end user.

V. Ghidra API Calls

For the purpose of tool development, the team made use of the flattened internal Ghidra APIs to easily access important scripting features. The FlatProgramAPI is a flattened version of the Ghidra main Program API that can be used for a variety of tasks, including reading information about and manipulating the currently loaded program, and prompting the user for input. Similarly, the FlatDecompilerAPI is a flattened version of the Ghidra Decompiler API that can be used to run the internal Ghidra decompiler and export the reconstructed program.

Primarily, a Ghidra script can be made extremely useful by taking user input so that each time the script is run, it can run with different values and configurations. As such, Ghidra provides quite a few different ways of requesting information or choices from the user via API calls through the FlatProgramAPI. The GhidraScript subclass of the FlatProgramAPI provides many useful methods beginning with the word "ask" that can be used to get user input via Ghidra's input GUI elements. The team uses these functions and more to conduct custom analysis of various binaries, as well as collect user input to aid in symbolic engine execution.

VI. External Engines

The tool uses a modular design that creates easily extensible code that can interface with other open source projects. As such, the tool is not only restricted to the Ghidra API, but can delegate tasks to other reverse engineering engines and frameworks as needed for better coverage and performance. Below, the team notes some engines that are already implemented into the tool, for ease of use.

A. *RISC-V Symbolic Execution*

The first implemented engine is the RISC-V Symbolic Execution Engine [2], created by Sice Squad, and is one of the main inspirations for the team's tool. It allows an end-user to symbolically execute RISC-V binaries by building on existing work from the Binary Ninja team and implementing internal support through Z3 [3] code.

The tool automates the initialization of Sice Squad's engine, by automatically inputting the code base address, file base address, and other parameters of the binary directly from Ghidra's FlatProgramAPI. The end-user is able to select a group of address to avoid or target, using the GUI to easily search for addresses from within Ghidra. Finally, the tool automatically runs Sice Squad's engine, which symbolically executes the program, and displays the results that were found.

B. *Z3 Theorem Prover*

The second implemented engine is the Z3 Theorem Prover [3], a cutting-edge SMT solver that can efficiently solve for arbitrary constraints. Users may select constraints that are intractable to solve by hand or difficult to comprehend and pass these constraints to Z3 for analysis.

As the team will discuss in the next section, the tool attempts to automatically identify constraints that may be placed upon an input to a program. If found, these constraints can be passed to Z3, which solves constraint-based problems orders of magnitude faster than typical symbolic execution engines can. Once Z3 is finished solving, the tool outputs any results that have been found, including a solution to the constraints if it exists.

VII. Custom Analysis

In addition to acting a shim to external engines, the tool was intended as a general aid to reverse engineering binaries. As such, the tool attempts to perform rudimentary analysis on a binary's metadata to identify points of interest for the end user. One implemented is the analysis of the Levenshtein distance between the binary's exported function names, colloquially known as “fuzzy string matching”. Since many generated functions and other functions of interest typically have similar names (i.e. “check123123” and “check342342”), the tool seeks out the functions with low Levenshtein distance between their function names and return them to the user. This is most efficiently done with the use of a BK-Tree, which is a data structure that is most commonly combined with Levenshtein distances to aid in spell-checking. Thus, it is possible to identify interesting clumps of functions that may contain clues on how to best reach the solution state.

The team then expands on these interesting functions by conducting some automated analysis using custom regular expressions that attempt to parse all elements of an if-else function. As a basic control flow function, analyzing conditional statements is the main goal of reverse engineering, as they provide the most information on how a program behaves. The implemented custom regular expressions use Ghidra’s internal FlatDecompilerAPI to retrieve a C representation of the binary, then automatically extract the constraints and feed them into an awaiting Z3 solver [3] instance. Z3, being a state-of-the-art SMT solver, attempts to find a solution for these constraints, and does so in a fraction of the time a typical symbolic execution engine takes, taking only 0.3 seconds instead of 10 minutes 27 seconds, a final speedup of 2090%.

VIII. Results and Discussion

This tool has been tested on a variety of challenge binaries in order to confirm its functionality. These tests focus on architectures that are targeted by the tool, specifically RISC-V and x86.

A. CSAW ESC 2020: Qualification

The RISC-V binary `qual-esc2020.elf` [7] served as the CSAW ESC 2020 qualification challenge for finals. It consists of three challenge functions, all of which were efficiently solved using the tool. See Table I. Finally, the team further verified the tool’s solutions by running them on the provided SiFive hardware.

TABLE I
Addresses for finding solutions to the CSAW ESC 2020 Qualification Binary

Challenge	Start Address	Find Address	Avoid Address
1	0x20400232	0x20400320	0x204002bc
2	0x2040032e	0x204003b2	0x20400390
3	0x2040052e	0x204005ba	0x20400598

B. CSAW ESC 2020: Parthenon

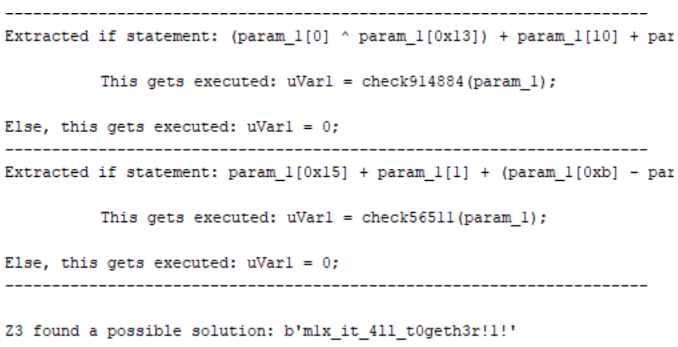
As expected, the tool’s internal symbolic execution engine is unable to solve every challenge problem. In certain cases where many paths exist, symbolic execution engines will encounter a phenomenon known as “path explosion”, where the number of possible paths increases exponentially, rendering symbolic execution computationally infeasible.

The Parthenon challenge (`parthenon.elf` [8]) is an example of a RISC-V binary in which path explosion occurs. In the Parthenon binary, the input is hashes using a custom cipher before being checked against user input. The use of this custom hash function results in path explosion when run through the tool, which point to other analysis methods being required to solve this binary. Indeed, the team notes that no existing solutions for Parthenon utilize symbolic execution, with many instead opting for manual reversing.

C. DiceCTF 2021: Babymix

The `babymix` [9] x86 binary from DiceCTF 2021 serves as a demonstration of the tool’s custom analysis and reversing capabilities. The binary implements many input-checker functions that verify whether a certain portion of the user’s input follow a particular constraint.

The tool uses the techniques detailed in previous sections to parse and analyze the functions of interest, which all follow the format of `check######`, where # represents a numeric digit. Thus, the tool’s implementation of Levenshtein distance can easily identify and parse these functions, then input the extracted constraints into Z3, as seen in Figure 2.



```
------------------------------------------------------------------
Extracted if statement: (param_1[0] ^ param_1[0x13]) + param_1[10] + par

          This gets executed: uVar1 = check914884(param_1);

Else, this gets executed: uVar1 = 0;
------------------------------------------------------------------
Extracted if statement: param_1[0x15] + param_1[1] + (param_1[0xb] - par

          This gets executed: uVar1 = check56511(param_1);

Else, this gets executed: uVar1 = 0;
------------------------------------------------------------------


Z3 found a possible solution: b'm1x_it_4ll_t0geth3r!1!'
```

Fig. 2. Example output from Babymix

Since the `babymix` binary is more rudimentary compared to other reverse engineering exercises, the tool performs spectacularly, quickly outputting a solution to the entire challenge.

IX. Future Work

Due to the extensible design of the tool, future directions for the tool include the inclusion of analysis of binaries compiled for the ARM architecture, which would be a valuable asset to the existing x86 and RISC-V toolbox due to the popularity of ARM CPUs in modern mobile and embedded environments. Additionally, the inclusion of popular symbolic execution engines like Angr [10] would open up symbolic execution past the RISC-V architecture to more common architectures like x86 and MIPS.

Finally, user interface improvements would greatly improve the accessibility of the tool, opening it to a wider audience of security researchers and enthusiasts.

X. Conclusion

In creating this tool, the team intends to make binary analysis and reverse engineering a much easier task, especially for binaries with vulnerable execution paths, such as ones used in Capture-the-Flag styled reverse engineering competitions. As such, the team implemented a custom script based upon the Ghidra Reverse Engineering framework that offers the following advantages over existing tools:

- Free and Open Source codebase, with supporting documentation to inform the creation of other similar tools
- Extensible framework to allow for future expansions based on various other analysis, fuzzing or testing engines
- Levenshtein-distance-based analysis to identify potentially interesting functions
- Regex-based constraint identification, parsing, and solving via Z3 to reverse engineer passwords for x86 binaries
- Symbolic execution based analysis and constraint solving via Z3 to reverse engineer passwords for RISC-V binaries

The team hopes that the creation of this tool will pave the way for further automated analysis of binaries, as well as greater ease-of-use between all of the major reverse engineering engines.

References

[1] "Ghidra," Available at https://github.com/NationalSecurityAgency/ghidra [Accessed 22 February 2021].

[2] "Risc-v symbolic execution," Available at https://github.com/sicesquad/riscv-sym [Accessed 22 February 2021].

[3] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[4] "Ghidra developer's guide," Available at https://github.com/NationalSecurityAgency/ghidra/blob/master/DevGuide.md [Accessed 22 February 2021].

[5] "Ghidra installation guide," Available at https://ghidra-sre.org/InstallationGuide.html [Accessed 19 February 2021].

[6] "Ghidra bridge," Available at https://github.com/justfoxing/ghidra_bridge [Accessed 14 April 2021].

[7] "Csaw embedded security challenge 2020 qualification binary," Available at https://github.gatech.edu/Embedded-System-Cyber-Security-VIP/ESCS-Hardware/blob/master/CSAW/csaw_esc_2020/qual-esc2020.elf [Accessed 14 April 2021].

[8] "Csaw embedded security challenge 2020 parthenon challenge binary," Available at https://github.gatech.edu/Embedded-System-Cyber-Security-VIP/ESCS-Hardware/blob/master/CSAW/csaw_esc_2020/challenges/setA/parthenon.elf [Accessed 14 April 2021].

[9] mmaekr, "Dicectf 2021 babymix challenge binary," Available at https://github.gatech.edu/Embedded-System-Cyber-Security-VIP/ESCS-Hardware/tree/master/CSAW/tools/babymix [Accessed 14 April 2021].

[10] 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," 2016.