Final Draft
Binary Function Fuzzer (BFF)
for CS5371 Soft Test for Mobile & Emb Sys

Jeremy Solmonson
School of Security Engineering
University of Colorado at Colorado Springs
Colorado Springs, CO 80922
Email: jsolmons@uccs.edu

***Abstract*—Vulnerability mitigation is essential for application development. Without the ability to timely find and mitigate software vulnerabilities, computer attacks will use the underlying weakness to take advantage of a computer system. Finding vulnerabilities as early as possible within the Software Development Life Cycle (SDLC), is the most cost efficient approach for application security. Unfortunately, most vulnerability analysis tools assess the software against the whole system, not individual parts of the program. As a result, the entire program must be completely written before the first vulnerability assessment is conducted. To integrate vulnerability identification earlier within the SDLC, we developed the Binary Function Fuzzer (BFF) tool. This tool allows software developers to integrate vulnerability identification within their unit tests and subsequently fixing the underlying issue. Additionally by bringing vulnerability identification to the individual developer enables faster mitigation strategies and stronger code ownership.**

I. Introduction

The process of reducing the vulnerabilities within software is extremely time intensive because programs are becoming larger and more complex. These complex program require extra scrutiny as the underlying logic may not be well known or well tested. Without through testing, vulnerabilities may be overlooked and released into the final program. Additionally, commercial software developers are rewarded for releasing their products as quickly as possible, which can limit the rigorous testing needed to reach hard-to-find vulnerabilities.

Exploitable vulnerabilities are extremely concerning for software development organizations. An exploitable vulnerability allows a computer attacker to use the underlying software in unintended methods. An attacker that can insert their own instructions as a substitute for the original programs is said to achieve Remote Code Execution (RCE). This can be considered the worst type of vulnerability as it allows the attack to execute their instructions up to the privileges of the application. For system services or processes that run with elevated permission, a RCE vulnerability allows the attacker complete control of the system. Testing for vulnerabilities on a continual basis reduces the potential for severely exploitable vulnerabilities.

To meet the organizational goals of balancing time with software quality, testing process is integrated into the earliest stages of software development known as unit tests. Unit tests allow the developer to ensure their features work as intended. However, unit tests do not usually include vulnerability testing. The problem with testing for vulnerabilities at the unit level is the time required to perform repetitive tests and observing unintended behavior. For large programs, execution from beginning to end can be severely time intensive.

To save time during testing, developers usually write units tests to ensure the successful functionality of their code. Writing tests that perform multiple failures is an after thought or non-existent practice. However, by writing tests that ensure a fault results in graceful program shutdown should be an integral part of software development. By augmenting the testing procedures within a unit level fuzzer allows the developers to identify vulnerabilities in a timely manner. In this paper, we discuss a new program called Binary Function Fuzzer (BFF), that can be used at the unit test level to search for vulnerabilities in individual units of code - see Figure 1. This allows security testing to be integrated at the lowest level of software development. By assisting software developers as they write the code, security vulnerabilities can be identified and fixed earlier within the SDLC. The final software product will have reduced vulnerabilities.

The problem with fuzzing unit inputs is their unreliability due to manipulations by previous blocks of code. For instance, if a program first checks the user input for non-ascii characters and ensures the length is less than a maximum amount, then these characteristics are followed into the subsequent execution blocks. A unit level vulnerability fuzzer should observe these constraints and inject inputs that are consistent with the input the program would receive.

Continually fuzzing inputs into individual functions poses environmental challenges. If the execution environment depends on specific variables, then the program needs to maintain the state of those variables in further testing. Take a file for instance, if the function enters with the current file pointer at a specific location within the file, the variable should returned to the same spot after the test completes. If the pointer is not returned, the future tests may be unreliable. This same logic is true for dynamically allocated memory, the current stack, and

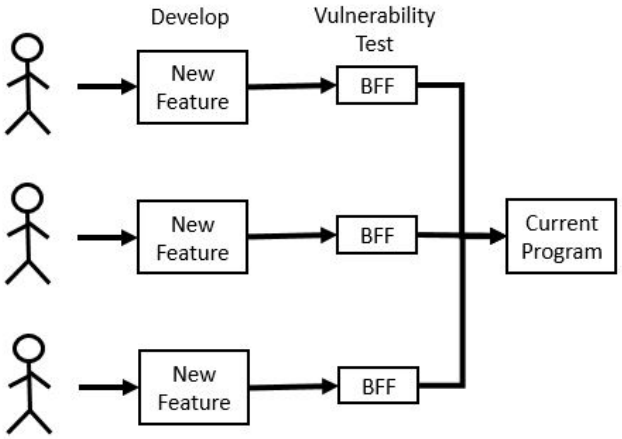

Fig. 1. Workflow with BFF

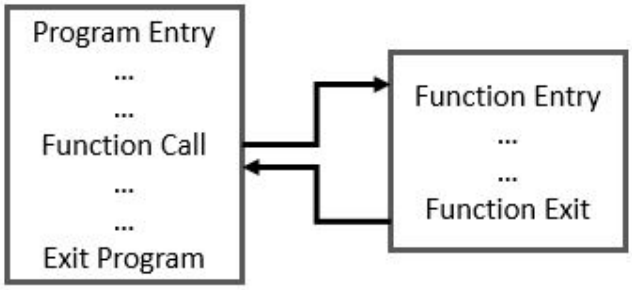

Fig. 2. Execution of a function

```
1 mov     rdi, rax
2 call    <vulnfunc>
```

Fig. 3. Binary representation of calling a function and setting argument (rdi)

registers. Any block level test needs to be aware of saving the non-input variables for further execution.

The BFF tool solve the above problems by providing a fuzzing capability that is integrated into unit level testing. This reduces the overall time required to test the system as the individual blocks of code are tested during development. Additionally, only the code up to the current function is analyzed. Alternative execution paths are not assessed. This allows the developer to better understand their integration into the current program. Further, by using a control flow graph with taint analysis, the BFF tool is able to identify input constraints and pass those to the subsequent fuzzer.

This paper provides the following contributions:

- Introduces the BFF program for fuzzing individual functions within the resulting binary
- A method that enables software developers to incorporate vulnerability identification within unit tests

II. Background

Below is a quick list of topics that is needed for background knowledge.

- Taint Analysis
- Function Call Interception
- Concolic testing / Symbolic Execution
- Control Flow Graph
- Dynamic Analysis
- Static Analysis

The main idea for this paper is to fuzz a select portion of a target program. By fuzzing a subset of the entire program, vulnerabilities can be identified in a deeper portion of the code. Usually fuzzers that locate hard to find, or deep, bugs can only run a few individual programs [cite]. The scalability of these tools can be overcome by analyzing the resulting file.

Implementing security testing with every phase of the SDLC creates resilient code. Unfortunately, the tools available to a development team to implement security coding practices are limited. This is especially true during the earlier phases of the SDLC. The ability to detect and fix a vulnerability is significantly cheaper if the problem is found earlier with the SDLC [cite]. By implementing security testing tools earlier with the development process, the overall program will be more resistant to vulnerabilities. Figure 1 demonstrates the workflow of implementing security testing after a new feature is developed, but before it is implemented into the current program. The workflow pushes security testing to the lowest level possible - the individual developer. By having the developer responsible for their own security testing, better code is developed. Further, the results are reviewed by the individual developer for their own feature. This allows the to learn from their past mistakes and take corrective action to prevent future vulnerabilities.

Functions are unique in the execution flow of a binary program due to the need to save the return address. When a function is called within a program, the return address of execution is saved for later use, the program saves the current states of the execution, sets arguments to the newly called function, and begins executing at the entry point of the function. Figure 2 visualizes the relationship between the original program and a function call. After the function is executed, the original program continues execution at the next instruction. A function is easily identified in binary analysis with the ”call” command - see figure 3. Once a function has completed its execution and is ready to exit, the reverse of the setup process is accomplished. The return value is saved into a register for the original execution to use. Additionally,

```
1 mov     eax,0x0
2 leave
3 ret
```

Fig. 4. Binary representation of leaving a function, setting the return value (rax), and setting the instruction pointer to restore execution (ret).

the return address is recalled to return execution prior to the function call and machine state is restored - see figure 4. This allows for a function to be considered a stand-alone execution and only relies on the original program for function arguments. This stand-alone properties allows for functions to be imported into other programs, and is compatible as long as the same type of arguments are supplied to the function. A fuzzer can import the same function and generate fuzz to supply to the function arguments. Doing so allows for individual features of the program to be fuzzed without the need to execute the entire program.

To abstract a portion of the program into a subset of the program requires Function Call Interception (FCI) [cite]. This allows a second program to use instrumentation by substituting, or modifying, the underlying main program. By modifying the main program, a developer can gain deep insight into a portion target program behavior. FCI allows the developer to substitute instructions or data variables to alter the main programs execution. While used for mostly debugging purposes, this can be extended to fuzzing.

Fuzzing is injecting semi-valid inputs into a target program with the intent to find bugs. This technique has been an area of recent research [cite] to fund unknown vulnerabilities. One of the problems with fuzzing is executing deep into the program to find hard to reach bugs. This issue has been partly solved with symbolic execution and constraint solvers[cite]. However, the problem still remains due to resource exhaustion. As programs become larger and more complex, more resources are needed to fuzz deeper into the program. One method to overcome this challenge is to allow the developer to identify areas of interest to fuzz. By fuzzing a portion of the program, the time needed to test can be significantly reduced. Further, this technique can be integrated into the software development process. Fuzzing a software feature as it becomes available will lead to higher quality coding practices.

To assist the developer further, taint analysis is able to identify user inputs and mark, or taint, them as untrusted. Following these variables within a program can identify execution paths that leads to user triggered bugs. The execution paths can be traced using static or dynamic analysis. Static analysis allows the tester to view the underlying program logic, commonly through a disassembler. The program logic can be further analyzed through a Control Flow Graph (CFG). The combination of static analysis and a CFG allows the user to view program execution through the basic blocks of code. As an analogy, the static analysis CFG view provide a road-map of the underlying program, while dynamic analysis drives down the road map. Dynamic analysis assesses a program during execution and provides relevant information back to the user. During dynamic analysis a program can be halted, modified by changing memory contents or registers, and then continue executing. This allows a user to assess different combinations to trigger function execution. By combining a control flow graph and tainting the input variables, a user is able to visually identify areas of potential concern within an application. Then using dynamic analysis, these input paths can be executed and observed in real time to provide further information.

The one issue with dynamic analysis is finding the right combination of inputs to execute a specific path. To overcome this challenge, symbolic execution with constraint solvers is used to identify inputs for path execution. By using a constraint solver to find the conditions for branch execution, a single execution path can be identified. This enables a repeatable process to narrow down a source of program feature. One tool in recent research that performs this activity is Angr [cite]. Angr is a binary analysis tool that is modifyable through the python programming interface. By adding to this exisitng tools, a program can be created to limit the scope of binary execution to a single path.

Combining the above techniques allows a software developer to fuzz their individual software feature.

III. Related Work

Fuzzing is an expansive topic that has received much attention over the past few years. [cite] While most research focuses on the performance of the fuzzer, that has been little research in new techniques that improve the fuzzing process. Angr was developed as a framework to stimulate new techniques in binary analysis. While Angr is a significant contributor to this paper, the research differs with a focus on fuzzing. Angr enables the function extraction through symbolic execution, while BFF focuses on fuzzing the results and reporting any vulnerabilities. Driller [cite] is a fuzzing tool that uses symbolic execution to identify hard to find vulnerabilities. It does this by symbolically executing the entire program. Ideally, BFF could be used as an add-on the the driller functionality. Concolic execution based on control-flow analysis [cite] has show significant progress in reducing the time to identify fuzzing inputs that leads to new execution paths. This feature is desirable when testing entire programs or individual functions. The inputs can be adjusted to the previous output and is considered the next phase into BFF development. Eliminating undesired inputs is essential in executing a fuzzer as efficiently as possible. TaintTrace flow tracing with dynamic binary rewrite [cite] has shown how tainting untrusted input can be used within binary analysis. While sanitizing untrusted input is a key defense within software applications, identifying the upper and lower bounds of the input provides great vale for fuzzing. By reducing the inputs to the minimal set of allowed values, allows the fuzzer to examine only valid inputs. A combination of static program analysis assisted with dynamic analysis will allow for further vulnerability discovery [cite]

IV. Experiment Design

The experiment is designed to compare full program fuzzers against feature specific fuzzers such as BFF. The result is to capture accuracy and efficiency of the fuzzer technology in terms of vulnerability identification and time of detection. By comparing the state-of-the-art fuzzers on these two metrics, we can find the overall performance increase (or decrease) each fuzzer exhibits. As there are different categories of various fuzzers, the experiment will focus on vulnerabilities

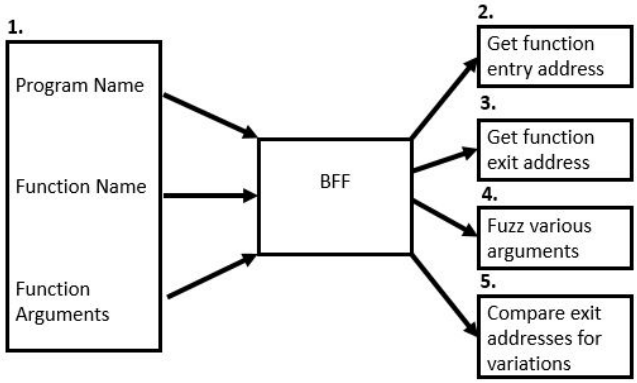

Fig. 5. BFF Concept

```
1  // Vulnerable Test function
2  int vulnfunc(char * test){
3    char vulnstr[16];
4    strcpy(vulnstr, test); // Vulnerability is here
5    return 0;
6  }
7
8  int main(int argc, char * argv[]) {
9    // call vulnerable function and return result
10   if(!vulnfunc(argv[1])) {
11       printf("success\n"); }
12   return 0;
13 }
```

Fig. 6. Vulnerable Code Example

that result in remote code execution. The fuzzers were selected based on age (updated in last 2 years), open source, run on x86_64 architecture, and able to execute Linux Executable and Linkable Format (ELF).

To identify a function with in a binary, static analysis plays a key roll by comparing the function name to it's address. To assist with this identification, a Control Flow Graph (CFG) is used to assist with analysis of the execution path. The CFG created by Angr was used as the function name to function address translation within BFF. By extrapolating the memory address within the application, we are able to save the program state prior to execution of the function. The current state of the program can be modified based on the user input. Using this feature in conjunction with Angr's function callable method, we are able to execute the function by providing function arguments. In essense, this has created a stand-alone program that only executes the function and records the results. As our tools is only interested in remote code execution, the state of the return address is most important. The downside of Angr is the use of symbolic execution for program flow. Symbolic execution substitutes concrete variables with symbolic variables to execute both execution paths. While ideal for larger programs, this features is not needed for this tool. However, the symbolic return address on a "good" execution remains constant. As a result, we an substitute the symbolic return address as the real return address. When the return address is overwritten, such as a buffer overflow, the program returns a different symbolic address. By comparing the "good" symbolic return address to the "bad" symbolic return address, we can conclude if a vulnerability exists.

BFF was designed to allow a developer to fuzz a specific function within a program - see figure 5. The developer would provide a file with function arguments to execute as the "fuzz" inputs (5.1). Based of a known good execution run, supplied by the developer, the BFF application would the get the function entry address (5.2) and function return address (5.3). The return address will be compared against all future runs. By repeatedly fuzzing various arguments within the function (5.4), the result my be different if the return address was incorrectly calculated. This would indicate an input was able to take control or modify the the return address. As a result, the return address is compared against the known good return value (5.5).

While we can conclude if a vulnerability exists within an application, none of the fuzzing tools can conclude if a vulnerability doesn't exist. While fuzzing has it's limitation, we can assess the false positive rate against reliable code. By testing secure code, we can determine if the fuzzer is prone to revealing false positives. False positives can be a result of a combination of various circumstances. While fuzzers should report almost no false positive, the test is necessary to gauge the effectiveness of the fuzzer. To thwart against a fuzzer that reports every application is vulnerable, we tested against known good programs. This two part process was to minimize the possibility of a false negative. The results of these tests are marked with asterisks within the results table. Each of the tests where the BFF tool identified a vulnerable application, a manual test was performed to verify the correctness of the found vulnerability.

A. Test Case

Figure 6 is a simple program that contains a buffer overflow. The design of the program is for the user to pass a string as a program parameter and the user supplied string is then copied to an in memory location. If the copy was successful, then a "success" message is printed to the screen. This simple program demonstrates taking control of return address by overwriting memory beyond the bounds of the allotted memory. Fuzzing this program with a traditional fuzzer would have the same effect as a function fuzzer such as BFF. The overhead to execute the function compared to the regular program is minimal. However, if we simulated additional code placed prior to lines lines 10 and lines 12, then fuzzing a function would be more advantageous.

V. Results

Need to perform the actual test

For each application, the following data points need to be recorded.

- Start time
- Time to first vulnerability identification
- Number of identified vulnerabilities

- Number of correctly identified vulnerabilities
- Number of incorrect identified vulnerabilities
- Number of non-identified vulnerabilities
- Sequence of input to trigger each vulnerability

VI. Threats to Validity

The BFF tool uses symbolic execution with a symbolic return address to determine if the function returned from a normal execution path. It's possible the symbolic execution engine decided to allocate an alternative memory location for execution. As a result, this would have a significant impact on the results. Fortunately, we did not see this within our tests, but this could happen on an alternative symbolic engine.

While similar tools were selected for comparable testing, some tools are designed to execute faster in different environments. The tools we selected against may not have been ideal for our chosen environment. If the tests were to run on a different architecture, compilers, or operating system, then the results may have significant variation.

VII. Future Work

The BFF tool only examined functions within a binary application. While directly applicable to customized sections within a software application, this has a broader impact including exported functions and shared libraries. Other applications may use similar features that can take advantage of fuzzing individual sections of code, such as start and stop ranges. By providing a start and stop range, the fuzzer is no longer limited to functions, instead it can fuzz anywhere within the target program. This provides more execution paths and unique scenarios to enhance security practices.

The BFF tool only provides basic types for argument inputs (strings, integers, doubles). The advanced types such as structures, pointers, and unions would be advantages as programmers become more abstracted. Providing this capability would require an enhancement to the fuzzing input file to supply the correct types to the function arguments. If a function argument is also the expected return result, then a memory range needs to be assigned to the program and stored after the function completes. The ability to process different types of data and functions can make fuzzing tools more universal.

To gain a higher fidelity of test results, modifying the fuzzing to tool to catch other types of vulnerabilities is desired. The BFF tool only looks for a specific type of vulnerability, remote code execution. Other vulnerabilities such as memory leaks and format string errors could be identified by examining the functions output. This can be tested against a broader category of NIST Juliet test suite to examine other vulnerable applications.

The BFF tool is a single threaded application and our goal was to test similiar tools. Most fuzzers that perform dynamic analysis have the option of utilizing additional cores of processing. The BFF tool can be upgraded to accept multiple instances of dynamic fuzzing.

VIII. Conclusion

The results from our test case proved the ability to fuzz individual components within a larger program. Breaking down a larger program into subcomponents provides scalability and ownership over the features within a given program. This advancement will allow developers to perform security tests as they develop code for their target program. The BFF tool provides a mechanism to fuzz individual functions which comprise of a much larger program. The speed at which the fuzzer increases performance is directly related to the execution time of the target program. By examining only areas of interest within the target program, we are able to utilize execution of specific targeted function. By introducing the BFF program we hope to inspire future fuzzing techniques that will assist with vulnerability prevention. Providing security tools to developers as they write their code will reduce the amount of vulnerabilities and improve secure coding practices.

References

[1] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 300–311. [Online]. Available: http://ieeexplore.ieee.org/document/7985671/

[2] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," vol. 67, no. 3, pp. 1199–1218.

[3] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," vol. 37, no. 6, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2382756.2382798

[4] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, pp. 94–105. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2931037.2931054

[5] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, pp. 559–570. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884853

[6] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling mutation testing for android apps," pp. 233–244. [Online]. Available: http://arxiv.org/abs/1707.09038

[7] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 33–44.

[8] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, pp. 643–653. [Online]. Available: http://ieeexplore.ieee.org/document/7985701/

[9] P. McAfee, M. Wiem Mkaouer, and D. E. Krutz, "CATE: Concolic android testing using java PathFinder for android applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, pp. 213–214. [Online]. Available: http://ieeexplore.ieee.org/document/7972811/

[10] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 1–11.

[11] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *11th IEEE Symposium on Computers and Communications (ISCC'06)*. IEEE, pp. 749–754. [Online]. Available: http://ieeexplore.ieee.org/document/1691114/