SCALABLE CONCOLIC EXECUTION OF COTS BINARIES WITH THE RIVER FRAMEWORK

Eduard Stăniloiu[1], Rareş Ştefan Epure[2], Răzvan Deaconescu[3], Răzvan Nițu[4], Răzvan Rughiniş[5]

Software systems increase in complexity as the time passes and new business requirements arise. Coupled with the fast pace and increasing demand of software development, this makes room for bugs and leads to a bigger attack surface. Fuzz testing and symbolic execution have proven to be successful methods to uncover untested paths and vulnerabilities, and have been adopted by teams across the globe. Software products regularly use 3rd party pre-compiled libraries and components. Those custom of-the-shelf (COTS) binaries make symbolic execution difficult as the fuzzer cannot insert the instrumentation code during compilation, thus having to resort to dynamic binary instrumentation. We present River, a concolic execution fuzzer for COTS binaries. We emulate the system-under-test (SUT) with Triton, a dynamic binary analysis library, and we delegate system and library calls to GDB. By doing so, we avoid path explosion and the need to manually implement system and library calls in the fuzzing framework, two problems that KLEE and angr have. We compare our solution with KLEE by testing on the libraries from Google FuzzBench. After running libarchive under the two for 30 minutes, River has a branch coverage of 4.56%, while KLEE has a 1.78% coverage; after 1h30 minutes, River has reached 6.06% compared to the 2.07% reached by KLEE.

Keywords: concolic execution, symbolic execution, COTS binaries, binary analysis, dynamic binary instrumentation

1. Introduction

Software security is essential to nowadays systems. The world has witnessed the rise of cybersecurity attacks in recent years, as depicted in Figure 1.

[1]PhD candidate, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: eduard.staniloiu@upb.ro

[2]Student, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: rares$_s$*tefan.epure@stud.acs.upb.ro*

[3]Lecturer, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: razvan.deaconescu@cs.pub.ro

[4]PhD candidate, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: razvan.nitu1305@upb.ro

[5]Professor, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: razvan.rughinis@cs.pub.ro

Exploits of in the wild vulnerabilities have cost companies billions [3], having researchers at the IBM System Science Institute estimate that fixing a vulnerability costs 100 times more than the development costs[8].

In fields such as IoT and automotive, auditing software security is challenging because the user receives a piece of hardware with a custom off-the-shelf (COTS) binary that runs on it. The user is forced to place his trust in the 3rd party, as he is not able to validate the software, since he does not have access to the source code.

Fuzz testing can help teams automate the testing procedure, and discover untested paths and vulnerabilities in their products and 3rd party components. If the source code is available, the fuzzing framework will add instrumentation code to the generated binary, which allows the framework to monitor the system under test (SUT) during runtime and improve the input generation phase of the fuzzer; this is known as *white-box fuzzing*. When there is no source code available, the fuzzer generates input (either from scratch or from an initial corpus) and checks the output (end state) of the SUT to determine if a crash has occured; this is known as *black-box fuzzing*. In-between the two methods is *gray-box fuzzing*: even though the source code is not available, the fuzzer will instrument the binary instructions in order to monitor system during runtime and guide the fuzzing process.

Static binary rewriting means being able to modify a compiled binary such that it remains executable after the changes. Static binary instrumentation would provide the best performance for a fuzzer, but binary rewriting is a hard, active research problem[32][10] [23][30][31] that has yet to have a sound, cross-architecture solution. Dynamic binary instrumentation (DBI) modifies the binary execution during runtime. This is a lot easier to achieve, as it does not depend on complex static analysis and it enables the user to leverage runtime information[19][6][5][28][11]. The drawback of using DBI is the runtime overhead that it incurs, which can range between 1.2x to 5x. When fuzzing, one desires the smallest instrumentation overhead, as that will allow the fuzzer to try more inputs in the same amount of time. We plan to compensate the overhead incurred by DBI by using symbolic execution.

We want to be able to perform efficient gray-box fuzzing on COTS binaries. The problem with black-box fuzzing is that you don't know anything about your target: as such, just testing against random input will not get one very far. Thus, we want to enhance the River fuzzer in order to perform symbolic execution on binaries through DBI: 1) trace the binary execution, 2) each time we encounter a branching condition, we save the encountered condition in the form of relations between symbolic variables, 3) the symbolic variables represent our path constraints and will build an algebraic equation of the current execution path, 4) so we can provide the equation to a satisfiability modulo theories (SMT) [9][4] solver, such as Z3[21], to discover if there are any input

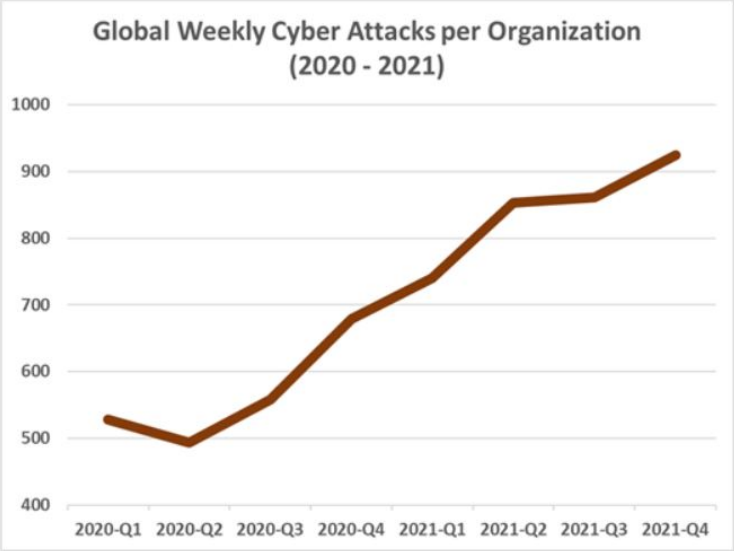

FIGURE 1. Rise of Global Weekly Cyber Attacks per Organization[1]

values that satisfy the constraints of a given program path. By using symbolic execution, one can quickly discover the "interesting" input values that improve the code coverage of a SUT and trigger vulnerabilities; this is because the constraints equation mathematically represents the different states that a regular fuzzer might discover by running multiple inputs.

Library and system function calls represent the biggest challenges of performing symbolic execution on binaries. Figure 2 presents a regular application flow, with all the transitions starting from user space and ending in kernel space. Starting with the program entry point instruction, we will iterate through the binary instructions until we reach a function call, eventually translated into a system or library function call.

Library function calls lead to a huge complicated constraints equation, known in the literature as *path explosion*[2]. The equation could undoubtedly become exponential as a consequence of the number of branches, which greatly increases the time required by the SMT solver to find equation solutions, to the point that it takes too long to be usable or it can not find a solution anymore. Even if we would have an ideal SMT solver that is not affected by the path explosion problem, system calls pose a new challenge: the implementation of the call resides in kernel space so the fuzzer cannot instrument those instructions as it does not have access to them. The solution is to either reimplement or emulate such calls.

Other symbolic execution fuzzers and frameworks, such as angr[29] and KLEE[7], reimplement common library functions (eg. `strlen, printf` etc.) and system calls (eg. `open, read, write` etc.) in order to circumvent the issues briefed above. There are two problems with this approach: 1) this does

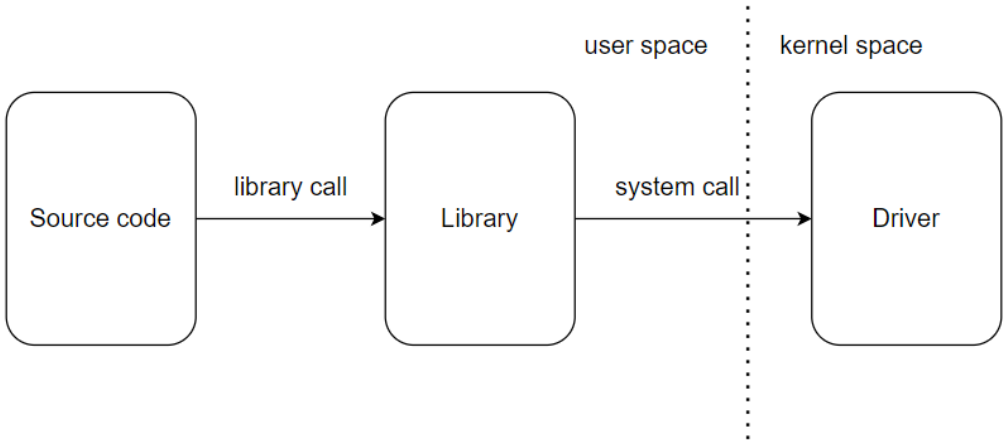

FIGURE 2. Application Flow: from User Space to Kernel Space

not scale as one cannot know all the functions that will be used by a user in his codebase, thus it requires extensive analysis to implement the missing stubs so one could fuzz the binary target and 2) the custom implementation might not behave exactly the same as the original library or system call, thus unwilling modifying the behaviour of the system under test.

Our proposed solution will use River to dynamically instrument only the instructions that are part of the analysed binary (no libraries or system calls). We integrate River with GDB so we can delegate the execution of library and system calls to it. Thus, River will only trace and symbolically execute the binary instructions of interest.

There are a series of aspects that we have to consider after the GDB executes the system or library calls:

- must maintain the consistency between memories of every process. The process from River has to have the same information even after the execution of GDB.
- must restore the function context after the execution of GDB.
- must keep the consistent symbolic state before and after a system or library function call.

The significant advantage that we have through our approach is that we emulate system and library function calls. Thus, the developer does not have the overhead associated with the implementation of every system call, nor does he have to do anything else in order to perform concolic execution. We claim that our implementation is scalable as it does not require the developer to debug the fuzzing tool and implement any missing utilities.

The remainder of this paper is organized as follows: Section 2 presents the background, Section **??** highlights the design of our collections and the problems encountered, Section 4 discusses the implementation and Section 5 presents the evaluation. We conclude with Section 6.

2. Background

2.1. The Architecture of The River Fuzzer

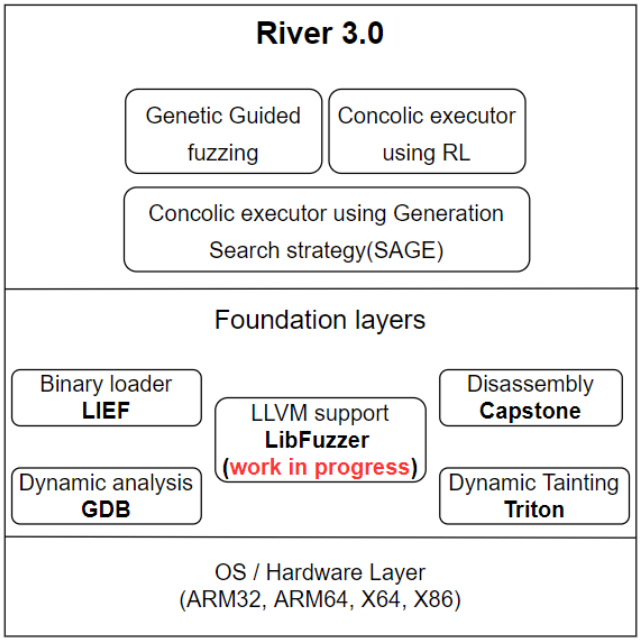

FIGURE 3. River Architecture

River is a cross-platform fuzzer that can run on all common operating systems and supports the following architectures: **X64, X86, ARM32** and **ARM64**. The fundamental components in its architecture are built using the following libraries:

- **LIEF**[17] serves at loading binary files. Using this tool, we can modify the executable file, access offsets, sections and the entry point, functionalities used in restoring the memory, and help to determine the position for necessary instructions.
- **Triton**[27] used in dynamic symbolic execution. Moreover, its usage in the River is to create an AST and make a dynamic taint analysis that verifies the security implications based on the flow generated by the user input. We can access functions from memory with this in River. This component has two substantial functionalities: the taint analysis function and the symbolic execution engine. Moreover, we can modify regions of memory and set symbolic expressions at different memory addresses.
- **Capstone**[26] is a component in the LIEF library because we can disassemble executable files through it. This is a high-performance framework, designed to provide the semantics of the disassembled instructions. Moreover, it is a thread-safe tool.

- **z3-solver**[21] is a theorem solver. Its usage is in the River and the Triton library to solve different constraints gathered from the branch instructions. The result after evaluating such expressions is a valid input that feeds the fuzzer in order to detect flaws in the system.
- **GDB**[13] is a tool through which we emulate the library and system calls and restore the context. We can inspect the memory at different moments from execution, using its capabilities of dynamic analysis. We can analyze the content of registers and acquire information about memory sections.

This framework supports three fuzzing methods, as you can see in Figure 3: concolic execution based on Generation Search strategy[14, 24], Genetic Guided fuzzing, and concolic execution using Reinforcement Learning[25, 24]. The similarity is that all of these methods generate paths based on the instructions from the executable files. Differences come from the way fuzzers choose paths. Genetic-guided fuzzing calculates traces based on a genetic algorithm, whereas the concolic execution using RL relies on an algorithm of Reinforcement Learning.

3. Methodology

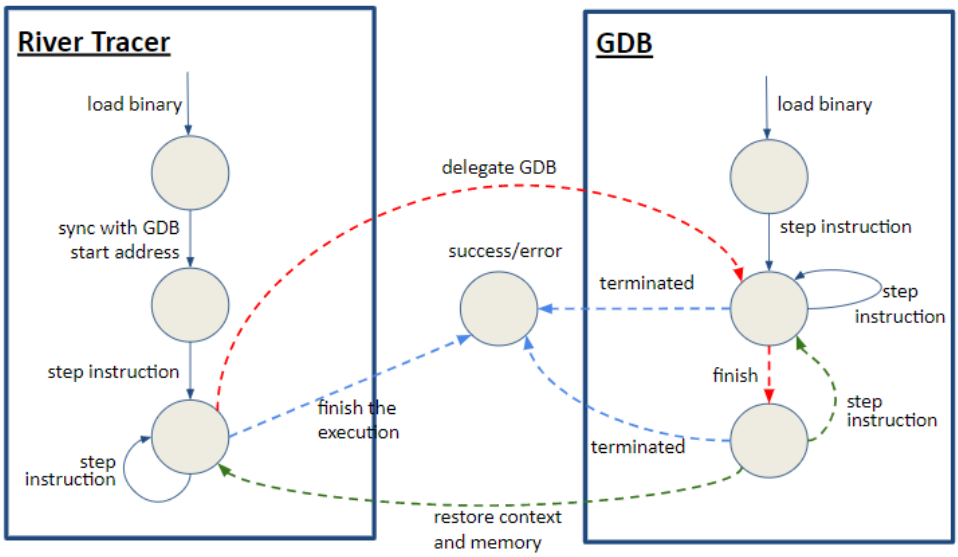

FIGURE 4. Control Flow

In Figure 4, we present how the system's components interact. We illustrate in the previous scheme the integration of GDB, which steps over system calls to reduce the number of states. Consequently, the size of the paths diminishes since the fuzzer omits to create the symbolic expression based on the branch instructions related to the system calls. The central point that we want to prove through this architectural scheme is the behavior of the fuzzer before and after a system or library function call.

The emulation method implies two processes, the River process, which builds the paths based on the instructions from binary files, and the GDB process, whose purpose is to emulate the entire functionality. Both start with loading the same binary file and mapping at the same addresses to ease the function of restoring the context. The system and library function calls have the implementation in kernel-space, and the River process cannot access it because this one runs in user-space. Therefore, GDB is the one that goes through instructions from these classes of calls.

The GDB process is the one that drives the execution of the River process, determining the flow, the latter updating its memory and context based on the information of the former after the GDB finished executing the system or library call. The dashed lines from Figure 4 are categorized based on colors, and each color illustrates that the actions from GDB and River are interconnected (performed simultaneously).

The first step after mapping the binary files into memory is to synchronize the program counter from River with the one from GDB. The executable files, compiled under some parameters, can go to such situations. The problem is that River always starts the execution from the tag of the entry point function, whereas GDB steps over the instructions correlated with the preamble of the entry point function in some cases.

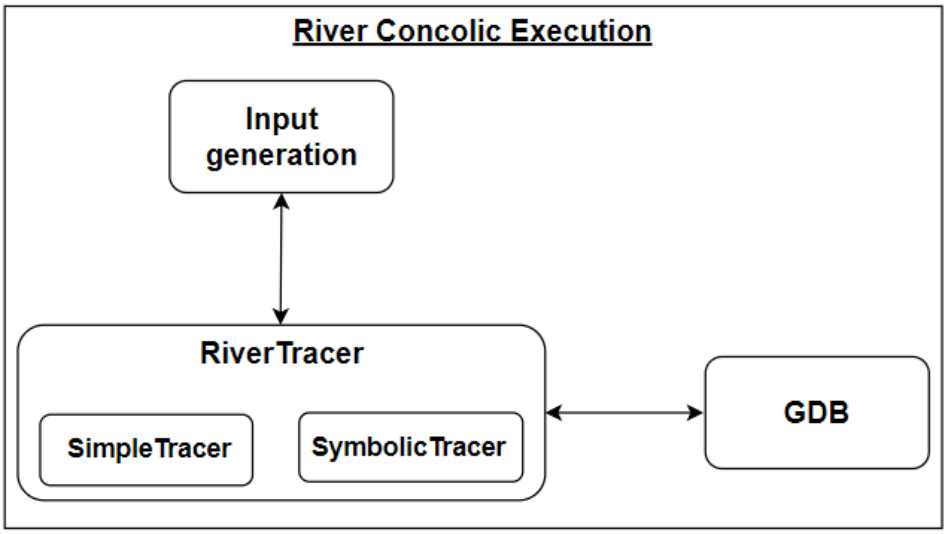

FIGURE 5. River Concolic Execution

The fundamental approach in River for fuzzing is the concolic execution, which represents a combination between concrete and symbolic execution. Concrete testing is inefficient in the context of real-world applications, which have numerous infeasible paths. Thus, the fuzzer uses the symbolic execution to determine tracks from the control flow graphs, which can cause rare errors. River implements these concepts in the RiverTracer through two submodules.

The purpose of the Simple Tracer module is to execute the binary file on a specified input. The result is a list of basic blocks representing the addresses of continuous instructions[24] until the execution has reached a jump directive. River computes a priority for the associated input in the input generation part based on the list of basic block information. The associated priority determines the following evaluated input.

The Symbolic Tracer functionality is similar to the Simple Tracer module, but the former differentiates from the latter by using the dynamic taint analysis. The system or the application runs with a symbolized input, which catches the jump conditions associated with the branch instructions along the trace, and these instructions bound the basic blocks. The outcome of this module is a Z3 serialized list of constraints passed to the Z3 solver in the input generation part in order to create more sets of test data.

4. **Implementation**

4.**1**. **River Tracer**

The initial approach in River was to use the Triton library to emulate the instructions. Although, there were limitations because of the lack of implementation in Triton of some directives. Moreover, River was not capable of loading all the libraries linked to the application. The proposed solution introduces the GDB module to remove the overhead caused by the necessity of adding these implementations. The purpose would be to solve one of the significant problems which occur in the current fuzzers. Our strategy is to emulate the instructions and system calls to remove the dependency between the fuzzer and them.

In the context of the symbolic execution, the associated control flow graph of the evaluated application would contain information about system or library calls in the AST representation. This sort of information is redundant in this situation because we would like to test just the application code. By delegating GDB to emulate these system or library calls, we will not add this kind of information to the control flow graph of the application (specifically from the internal AST representation from the Triton library). As a result, we will reduce the path explosion that is a real problem in the current fuzzers. We will integrate the described solution in a tracer that will make symbolic and concrete execution, creating a constraint expression based on jump, branch or call instructions.

4.**1**.1. *River emulation process.* Symbolic execution is the technique used by input generation, which obtains information from the control flow graph's traces. However, the performance of the fuzzer can decrease due to the associated drawback of this approach, named path explosion. The need for emulation comes from this necessity, in the case of the library and system calls, which could exponentially increase the depth of the graph.

Figure 6 shows the steps of the symbolic execution, the states representing the flow of the process. Every state contains a scope with concrete or symbolic variables and the following line of code of the application. As mentioned above, the result is a constraint expression calculated relying on the jump, branch, and call instructions[2]. At the end of the execution, the fuzzer evaluates the list of constraints and generates more inputs.

Having a deep insight into the method the fuzzer creates the constraint path, we can generalize a state of the execution in three components: a statement, a symbolic store and a partial path of constraints. The statement is the one that must be evaluated, usually associated with an assignment, a jump or a branch instruction. We can associate the symbolic store with a scope that contains program variables related to symbolic or concrete expressions. The evaluation of the statement comprises a series of aspects. In the case of an assignment, the symbolic engine will associate a variable with a value in the symbolic store for either symbolic or concrete variables. The branch instruction will create a constraint, attaching it to the list of constraint paths. The execution will continue on the appropriate branch based on the input. In the case of occurring a jump instruction, this will result in moving forward to the following statement.

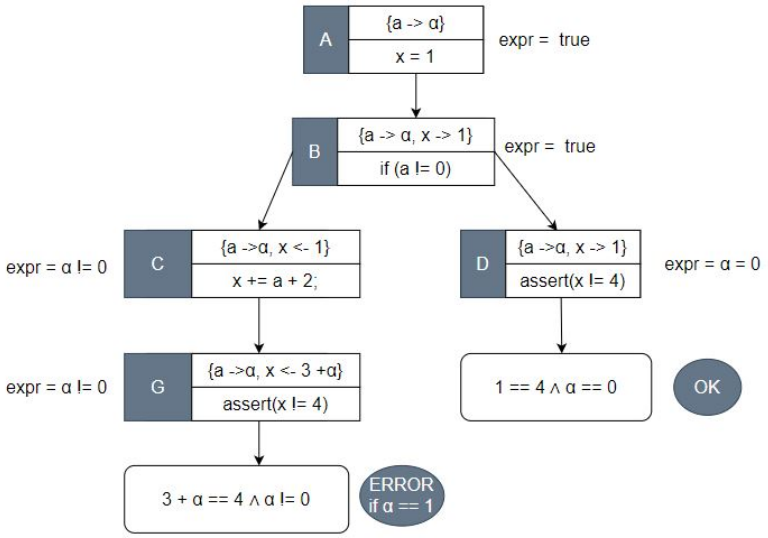

FIGURE 6. Symbolic execution graph

In the proposed solution, we load the executable file into memory twice: once by the River process and once by the GDB process. The latter will drive the execution. The reason for which these two distinct processes emulate the same instructions is that the River process builds the constraint path in the process of symbolic execution, whereas the GDB process emulates system and

library calls skipped by the River process. Both processes execute the same instructions only if these are in the mappings of the executable file.

River process loads the executable file and can access only the instructions inside the binary file, whereas GDB can access the instructions from libraries, not just from the binary file. In this situation, only the GDB will have access to the instructions from the library and system calls. Thus, GDB will have exclusive control of the flow when the River process wants to access a directive from a library object that the latter does not map. The River process will copy the memory and the registers from the GDB process address space into its own address space.

Every time a system or a library call occurs, the River process will update its memory based on the information from the GDB process. Because the size of the sections from the binary file can increase after a system or library call, the River will map into its memory the same size as it is in the memory of the GDB process. Examples, in this case, could be the heap or the stack because they could expand their sizes.

4.**1**.2. *Fuzzing loop.* River contains two significant components: tracers and an input generation part. The tracers are the ones that perform either symbolic or concrete execution. After a symbolic execution trace, the fuzzer obtains an expression based on the constraints paths, built on jumps or conditional branches, applied to symbolic variables. Input generation utilizes a seed as input in order to determine runtime constraints paths, used in generating more inputs. River works with one symbolic tracer and other simple tracers, which will execute the binary file to determine possible flaws in the execution. Thus, it will result in vaster code coverage based on constraints discovered in the symbolic execution phase.

The fuzzer makes dynamic taint analysis through Triton, which symbolizes the input for symbolic execution, and keeps track of instructions that interacts with the symbolized addresses or registers. Besides, Triton creates an AST based on the directives processed during the symbolic trace. After the first run, the fuzzer will execute the program using the generated input, trying to determine possible errors in the code. To create this loop in the target executable file should be defined a dummy array through which GDB and River will insert the input. In order to maintain consistency, we must place the data into memory at the same addresses in both processes. The name for this variable must be `inputBuf`.

4.**1**.3. *Maintaining consistency between components.* Context restoration is a vital part of the current implementation because River is the one that builds the symbolic expression. Thus, the fuzzer must be aware of any change in the application. The GDB is the component that emulates the behavior of library and system calls, and River waits for updates after the execution of such calls. The importance of context restoration comes from the desynchronization

between GDB and River. The memory and the registers from the fuzzer will be behind the state of the GDB process after a system or library call. So, they must synchronize because of the need for the fuzzer to create the expression of constraints and keep its condition up-to-date.

After a library or system call, River updates its registers by copying the information from the registers of the GDB process. However, the fuzzer modifies only those that changed their information after such calls. The River made the restoration in this way since the fuzzer must preserve the symbolic state. When River overwrites the content of a register after a system or library call, in this case, the fuzzer removes the symbolic reference to that register. Triton is the component library that shapes this behavior, as River executes symbolic execution through it.

The memory update goes on the same logic as the register restoration, reasoning identical. However, this approach comes with a drawback because we must iterate through the entire memory segments that contain permissions of writing to check whether a change occurred. Thus, the memory becomes the bottleneck of the fuzzer because River will search for modified values at different memory addresses after the system and library call.

The GDB Python API provides a way of controlling the memory and the flow of the program through an object called inferior, which is a program executed under the GDB. River accesses the memory through the Triton API, called `TritonContext`. Moreover, we can access information about the program during the run, such as memory sections, process id and others.

Using the functionalities provided by GDB Python API, we have the possibility to read the entire section of memory, which results in a series of bytes. The fuzzer compares the area from River with the one from GDB to speed up the execution because, in the case of equality, this one skips the following operations and moves to the next section. However, it is unlikely that a system or a library call will change the entire segment. Thus, we slice the memory in chunks no more than the cache size for an extra speedup. If the slice has the same values in both processes, the fuzzer will skip it. This comparison is possible since both tools, River and GDB, provide the memory content in bytes, making this operation efficient.

The problem of maintaining consistency between symbolic and concrete states has as a starting point the Triton library, which keeps a record of both states and creates the constraints paths based on this information. When River updates a memory value after a system or library call, Triton will concretize the content from that address. Thus, this library removes the symbolic information related to that address because the input generation part will not have the probability of influencing the behavior starting with that point of execution. Furthermore, Triton automatically concretizes areas of memory after River calls a set function on it. The memory update strategy relies on a dictionary of memory sections. The fuzzer updates the information from this dictionary

after every system or library call because some segments can increase their initial sizes. For instance, after repeated `mmap` calls, the heap segment can expand its size. So, to maintain consistency, River regenerates this dictionary of memory segments based on the information from the GDB process.

5. Evaluation

In order to validate our solution, we evaluated a series of applications that try to meet real-world requirements. We used a HTTP parser[22] (Figure 7) and an application that tries to parse a JSON object, named Fuzzgoat[12] (Figure 8). We choose the former because it contain regular system and library function calls met in every program, and the latter because it is intended to be used as a testbench for fuzzers. Another reason is that the fuzzer covers a significant part of the code in a reasonable amount of time despite the overhead caused by the emulation.

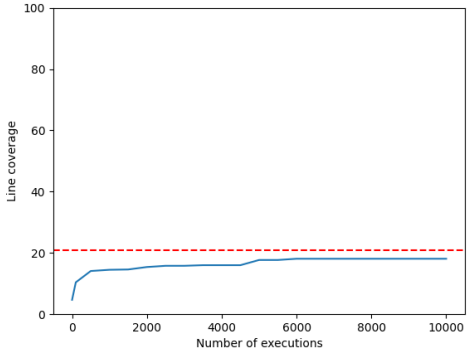

FIGURE 7. Line coverage for http parser test

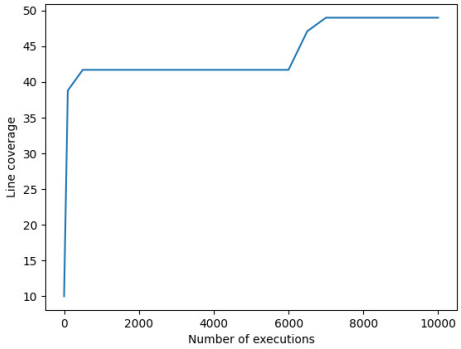

FIGURE 8. Line coverage for Fuzzgoat test

The HTTP parser has two components: the parser for the request and one for the URL. The line coverage for this library is that the URL parser has 21%, whereas the other component has 79%. We choose to test the URL parser since the input is not highly structured as it is for HTTP request parse. To support highly structured inputs in a reliable amount of time, River has to generate the data tests based on grammar. The current approach creates inputs relying on symbolic execution, which will take much more time to build suitable inputs.

5.1. Metrics and performances

So far we have evaluated that our functionality requirements are successfully met by our solution. Next, we compare our approach with similar ones from Klee. We used the Libre [16] and Libarchive [18] libraries from Google's

Fuzzbench [15, 20]. We also used Fuzzgoat, previously mentioned, in our data set. The results are shown in the following tables:

TIME	RIVER	KLEE
30min	49%	65%
1h	49%	73%
1h30min	49%	76%

tableBranch coverage for Fuzzgoat example.

TIME	RIVER	KLEE
30min	4.56%	1.78%
1h	6.02%	1.92%
1h30min	6.06%	2.07%

tableBranch coverage for Libarchive example.

TIME	RIVER	KLEE
30min	28.9%	fail
1h	28.9%	fail
1h30min	28.9%	fail

TABLE 1. Line coverage for Libre example.

Our metric to compare both tools is branch coverage because we can get only the branch and instruction coverage, whereas we have the possibility to analyze the line and branch coverage in River since the latter tests the coverage using the gcov tool and the former has an internal mechanism that makes the measurements. So, the shared metric is the branch coverage.

We can notice from the above results that Klee has a large coverage for the Fuzzgoat example, which contains regular functions and library calls. The other two examples are more complex in terms of system and library calls. Moreover, these applications imply a substantial number of such calls. Klee fuzzer fails with a SEGFAULT signal for Libre example because of the lack of implementation for an inline assembly call from the `valgrind.cc` file. The error occurs even though the Klee was compiled with support for libc++. A possible reason is the lack of some implementation for some system or library function calls. River tries to avoid this issue by emulation such calls through GDB. We can observe from Table 5.**1** that Klee is slower when the application involves a notable number of system and library function calls. Moreover, the Libre example implies a significant number of system calls from POSIX. Even though Klee has support for such calls, their approach implies a substantial overhead for them.

6. **Conclusions**

We have successfully added support for symbolic execution for COTS binaries in the River fuzzer. By delegating the execution of syscalls and library functions to GDB, we significantly reduce the risc of triggering a path explosion. Moreover, the associated overhead with the reimplementation of such calls drastically diminishes. Moreover, through this approach, we want to underline the possibility of emulating the system and library function calls by keeping the concrete and symbolic states consistent.

Restoring the symbolic state is a core aspect of the current approach, since the fuzzer can otherwise miss specific symbolized values after system or library function calls. To prevent any errors, we have an "aggressive" approach, restoring the memory and symbolic contexts after each GDB delegated call. The restore policy could be relaxed in order to obtain better runtime performance.

6.1. Future performance improvements

Our context restoration solution is fully implemented, which is crucial for our approach. However, the current implementation is very cautious and copies large sections of memory from GDB into the River process each time a system or library call is made in order to update the context in the fuzzer. This causes the memory to become a bottleneck because the performance of the tool is directly tied to the size and number of memory sections being copied. In addition to optimizing the use of memory, the testing tool could also benefit from using the functionalities of GDB and leveraging GDB's signal handler to improve the input generation process by filtering inputs based on previous data tests that have caused errors in the system.

One important factor to consider is how the fuzzer generates its inputs because if it uses the same path multiple times, it can lead to a decrease in performance. To improve performance, the fuzzer could avoid generating redundant data tests by altering its approach for input generation. Optimizing this component can lead to a reduction in the amount of time needed for testing. Thus, the fuzzer could achieve a wider code coverage in the same amount of time by adjusting its input generation process.

6.2. Future emulation improvements

During emulation, the fuzzer aims to guide the concrete execution to achieve the correct execution. However, as discussed throughout this chapter, the fuzzer also has an internal representation of the symbolic state, which GDB is not aware of because its purpose is only to execute the application. In most cases, the fuzzer does not need to consider the symbolic state when handling system and library function calls because it does not affect other addresses. This changes when the library calls in case should operate on symbolic memory, as it must be taken into account by taint analysis. For example, after a `memcpy` call, the destination address should have the same symbolic information as the source address, but the symbolic state of the destination address will not be updated by GDB because it cannot interact with symbolized information. In this regard, the current solution relies on hooks of well known, memory altering, functions, but this limits support. To improve this, we must design a more generic approach to handle these calls that require working with the symbolic state.

Acknowledgment

This work was supported by a grant from the Romanian Ministry of Research, Innovation and Digitization UEFISCDI no. 401PED/2020, and a grant from the Romanian Ministry of European Investments and Projects through the Human Capital Sectoral Operational Program 2014-2020, Contract nr. 62461/03.06.2022, SMIS code 153735.

REFERENCES

[1] Check point research: Cyber attacks increased 50% year over year - check point software. https://blog.checkpoint.com/2022/01/10/check-point-research-cyber-attacks-increased-50-year-over-year/. Accessed: 2021-10-20.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[3] Adam Bannister. Substandard software costs us economy $2tn through security flaws, legacy systems, abandoned projects. *URL: https://portswigger.net/daily-swig/substandard-software-costs-us-economy-2tn-through-security-flaws-legacy-systems-abandoned-projects*, 2021.

[4] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of model checking*, pages 305–343. Springer, 2018.

[5] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[6] Derek Bruening and Qin Zhao. Building dynamic tools with dynamorio on x86 and arm. 2017.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] Maurice Dawson, Darrell Norman Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3(6):49–53, 2010.

[9] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[10] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[12] fuzzstati0n/fuzzgoat: A vulnerable c program for testing fuzzers. https://github.com/fuzzstati0n/fuzzgoat. Accessed: 2021-10-20.

[13] Gdb: The gnu project debugger. https://www.sourceware.org/gdb/documentation/. Accessed: 2021-10-20.

[14] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[15] Fuzzbench: Fuzzer benchmarking as a service. https://google.github.io/fuzzbench/. Accessed: 2021-10-20.

[16] google/re: Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. it is a c++ library. https://github.com/google/re2. Accessed: 2021-10-20.

[17] Lief : Library to instrument executable formats. https://lief-project.github.io/doc/latest/intro.html. Accessed: 2021-10-20.

[18] libarchive/libarchive: Multi-format archive and compression library. https://github.com/libarchive/libarchive. Accessed: 2021-10-20.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[20] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[22] nodejs/http-parser: Http request/response parser for messages written in c. https://github.com/nodejs/http-parser. Accessed: 2021-10-20.

[23] Pádraig O'sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *Ifip International Information Security Conference*, pages 154–172. Springer, 2011.

[24] Ciprian Paduraru, Bogdan Ghimis, and Alin Stefanescu. Riverconc: An open-source concolic execution engine for x86 binaries. In *ICSOFT*, pages 529–536, 2020.

[25] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. Riverfuzzrl-an open-source tool to experiment with reinforcement learning for fuzzing. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435. IEEE, 2021.

[26] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 5(2):3–8, 2014.

[27] Florent Saudel and Jonathan Salwan. Triton: Concolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.

[28] Nathan Voss. afl-unicorn: Fuzzing arbitrary binary code. *Hacker Noon*, 2017.

[29] Fish Wang and Yan Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[30] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[31] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[32] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.