RIVER: A Binary Analysis Framework using Symbolic Execution and Reversible x86 Instructions

Teodor Stoenescu[1], Alin Stefanescu[2], Sorina Predut[2], and Florentin Ipate[2]

[1] Bitdefender, Romania
[2] University of Bucharest, Romania

Abstract. We present a binary analysis framework based on symbolic execution with the distinguishing capability to execute stepwise forward and also backward through the execution tree. It was developed internally at Bitdefender and code-named RIVER. The framework provides components such as a taint engine, a dynamic symbolic execution engine, and integration with Z3 for constraint solving.

1 Introduction

Given the nowadays extreme interconnectivity between multiple systems, networks and (big) data pools, the field of cybersecurity is a vitally important aspect, for which concentrated efforts and resources are invested. To mention only two recent examples in this direction, European Union just launched a new public-private partnership on cybersecurity which is expected to trigger a €1.8 billion of investment by 2020 [1] in advanced research and cooperation to improve the defence against the myriad of security attacks, and US currently organises, through DARPA, a cybersecurity grand challenge (CGC) [2], where successful teams compete to analyse and fix a benchmark of binary files using a combination of dynamic and static analysis, concolic, and fuzz testing.

Almost all the tools on the security market which aim to detect vulnerabilities of source or binary code employ static analysis or, more rarely, dynamic analysis through random values, a technique called fuzz testing. This may be more efficient than the alternative of symbolic execution that we explore here, but can miss many deeper or more insidious security issues. Symbolic execution is a promising approach whose foundational principles were laid thirty years ago [3], but which only recently started to regain attention from the research community due to advancement in constraint solving, various combinations of concrete and symbolic execution, and more computing power to fight the usual state explosion problem [4]. The basic idea of symbolic execution is to mark (some of) the program variables as symbolic rather than concrete and execute the program symbolically by accumulating constraints on those variables along the different paths explored in the execution tree.

Most of the symbolic execution tools work on source code or bytecode [5–7] rather than binary code [8–10]. However, binary code analysis is a very difficult

task due to its complexity and lower level constructs. On the other hand, it is better to run the analysis directly at binary level, because this is the code which is executed by the operating system. Moreover, in cybersecurity, usually only the binary file is available, so recent research efforts are invested into dynamic analysis of binary files [2] with companies such as Bitdefender joining the trend.

Bitdefender is a Romanian software security company and the creator of one of the world's fastest and most effective lines of internationally certified security software and award-winning protection since 2001 [11]. Today, Bitdefender secures the digital experience of 500 million home and corporate users across the globe and, for that, Bitdefender is constantly performing research activities in the software security area. The RIVER framework is an example of such internal research effort with 2 person-years invested in the project until now.

Contributions: The main differentiator of RIVER is the design and implementation of a set of extended reversible x86 instructions, which allows an efficient control of the execution and their integration into a symbolic execution framework. For that, the following artifacts were created: RIVER intermediate representation, which adds necessary and sufficient information to the x86 set of instructions in order to efficiently "undo" the operations when needed or to track certain variables as tainted; dedicated taint analysis and symbolic execution engines based on the above; and, as a byproduct, a debugger at binary level with forward and backward step execution capabilities.
A technical report on RIVER is online at: `http://tinyurl.com/river-tr-2016`

2 Description of the framework

This section details the overall design of the RIVER framework, which is shown in Fig. 1. RIVER (spelled backwards) stands for the *"REVersible Intermediate Representation"*. RIVER has a fixed length extended x86 instruction set and was designed to be efficiently translated to and from x86 normal ("forward") instructions. Its main novelty is the introduction of reverse ("backward") instructions. Also, specific tracking instructions were added to enable the taint analysis. This intermediate representation is depicted in the left hand side of Fig. 1. The RIVER code is obtained from an input as x86 native binary code (see bottom-left corner of Fig. 1) through the dynamic binary instrumentation component, by means of disassembly. Then, modified code is used by the components for on-the-fly reversible execution and taint analysis. All these are used by the symbolic execution engine which also uses a state-of-the-art SMT solver, Z3 [12], for dealing with the constraints for the symbolic variables (see top of Fig. 1) but also on-demand snapshots to save certain memory states. All these and various other aspects are discussed in this section.

RIVER intermediate language Now we describe RIVER intermediate language (IL) by presenting a couple of design choices. More details and an example is given in the RIVER technical report mentioned above. First of all, RIVER code is obtained automatically from the input native x86 through the dynamic

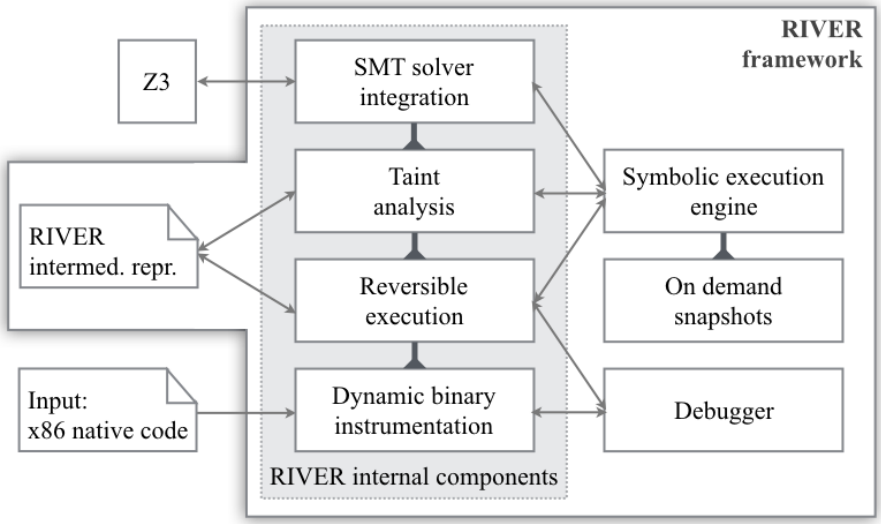

Fig. 1. RIVER architecture

binary instrumentation component (DBI) which is plugged in the reversible execution component (see bottom-left corner of Fig. 1). Thus, RIVER augments translated code in order to make it reversible. It uses a shadow stack in order to save instruction operands that are about to be destroyed. The original instructions are prefixed with operand saving ones. DBI also generates code for reversing the execution so that the destroyed values can be restored from the shadow stack. The RIVER instructions include modifiers, specifiers, operator codes and types as well as flags and a special field for the family of the instruction. These additional information is used to identify the prefixes and operand types and registers of the original instructions and help the data flow analysis.

RIVER DBI component also contains its own disassembler, which augments the code with the following properties: (a) implicit operands: some instructions implicitly modify registers and memory locations. These are added to the instruction as implicit operands; (b) register versioning: in order to simplify data flow analysis, the disassembler versions every register use; (c) meta operations: since the x86 instruction set is not orthogonal, some instructions may be split into several sub-operations, and (d) absolute jump addresses: relative jump operations are augmented with an additional operand containing the original instruction address. This makes it easier to compute the jump destination.

Reversible execution component The reversible execution engine (see middle of Fig. 1) enables the forward and backward control of RIVER IL code that was translated from the native x86 code through the DBI component. It operates at the basic block level, i.e., a sequence of instructions terminated by a jump, by replacing the jump instruction in order to maintain the execution control. To implement reversibility, the RIVER translator inserts RIVER-specific instruc-

tions in the translated code. Then, the RIVER translator generates a second basic block for reversing the effects of the first block.

Based on the above, we developed a forward and backward binary debugger (see bottom-right of Fig. 1). We created it to be used by the software developers and security experts at Bitdefender, who need to examine dynamically certain behaviours of binary files with a fine-grained control. It operates at basic block level and it has a web front-end using JavaScript bindings for RIVER. Moreover, it offers the possibility to set breakpoints, but also so-called "waypoints", which are similar to breakpoints but referring to points in the past of the execution.

Taint analysis component This component records the spread of taint through a program which uses tainted values. We implement classic taint spreading algorithms, but we adapt them to our RIVER IL to take into account also the reversibility feature. Technically, we added tracking instructions in RIVER IL which are used by DBI to enable determining locations (both memory and registers) that have been directly influenced by the input values. Initially all input locations are marked as tainted and everything else is untainted. At runtime, any instruction having a tainted operand produces tainted results (with some exceptions). There are two ways of tracking locations: using simple boolean values or binding custom values to memory locations (pointers to symbolic expressions). We use the former for simple taint analysis (if used as standalone) and the latter for symbolic execution.

Symbolic execution engine In order to perform various types of analysis and testing using dynamic symbolic execution, the program has to exercise a large set of paths through its execution tree. The more paths are explored, the higher the coverage of examined behaviours. However, since the enumeration of paths is computationally expensive, several approaches have been proposed to minimize its footprint [4]. Our symbolic execution engine (see right of Fig. 1) aims to tackle the path explosion problem through its distinctive feature of reversibility. More precisely, instead of re-executing paths from the beginning each time, we generate them through backtracking (using, e.g., a depth first search strategy) and use the reversibility to keep the memory usage low for the backtracking steps. Moreover, we keep only the current path in memory rather than a whole set of paths and snapshots. Thus, we try to exploit the temporal and spatial data locality, since most execution paths have a lot of common subsequences. We do the above by keeping track of two things in parallel: a concrete stack for the current path plus, only when needed, snapshots. We optimise the latter during reverse execution using many implicit micro-snapshots as opposed to (expensive) macro-snapshots usually used by the current symbolic execution approaches. The micro-snapshots keep only the modified memory locations, so we can easily restore the previous snapshot at each program point.

It is a high priority for us to keep the snapshots at a minimum, and use it only on demand, i.e., when we cannot reverse the execution of specific instructions, such as system calls, processor exceptions, or interrupts (e.g., "`0x2e`").

The fact that they are quite uncommon also helps our performance. Furthermore, we try to avoid also the snapshots associated to system calls: we have started a detailed analysis of the reversibility of these problematic functions, by systematically examining Windows Native API (NTDLL) and implementing their inverse functions, whenever possible.

Regarding the symbolic execution engine, we do not implement "pure" symbolic execution, but use concolic execution, i.e., mixing concrete and symbolic execution at the binary level. Thus, instead of being only symbolic, the inputs have a concrete value which is a representative of the symbolic domain. Besides, the taint analysis component tracks the symbolic values.

Other technical aspects RIVER framework is written in C++, having 14 KLOC in the current stable version, but is still under further development, with more components, optimisation and types of analyses to be added soon.

RIVER IL currently covers about 87% of the integer x86 instruction set, which is the core of x86. This percentage is high enough to run most binary programs in RIVER reversible mode (including specific debugging) and for taint analysis. However, we cannot compare yet the performance of RIVER with other frameworks using symbolic execution on binaries, because the SMT solver integration does not have a high enough coverage to run on existing benchmarks.

Also, the symbolic execution engine implements only a straightforward depth-first exploration of paths using the C APIs of the RIVER components in the middle of Fig. 1, but we are now adapting several advanced features available in other state of the art symbolic execution frameworks [6, 13, 9, 3, 14, 10, 2]. There is great advancement in the dynamic symbolic execution research community, which was increasingly active over the last decade [4, 3].

Moreover, we now develop an integration of our concolic execution with a parallel fuzz testing module, in order to increase the path coverage. We designed a distributed processing framework based on Apache Spark and Hadoop to apply fuzz testing on several parallel machines and obtain a first test suite with a good coverage. Then, we apply symbolic execution by tweaking certain paths to increase coverage, as done also by others [15, 16]. This is still work in progress.

Also, RIVER IL increases sixfold the size of original x86 code. To lower this overhead, we are currently implementing some classic code optimisation methods such as instruction reordering. After first experiments, we estimate to reduce the size of RIVER code to only double the size of the original code, which should be an acceptable trade-off.

3 Conclusions

In this paper, we presented RIVER, a new binary analysis framework built from scratch with the idea of reversible basic block at its core. RIVER has all the components needed to perform dynamic symbolic execution, including: dynamic binary instrumentation and reversible execution, which enabled the construction of a dedicated debugger, and also, taint analysis and SMT solver integration,

which enabled a lightweight symbolic execution engine with minimized footprint. This architecture was based on a novel intermediate representation, RIVER IL.

We plan to use RIVER internally at Bitdefender in order to both extensively test our commercial products, but also to find security vulnerabilities in external binary files, which is Bitdefender's core business. To reach this level, we need to implement several improvements mentioned before and then tune the framework for certain types of vulnerabilities. This will be our focus for the next months. Moreover, we want to experiment with idea cross-pollination between RIVER and related tools in both directions, i.e., to implement in RIVER heuristics that proved efficient in other frameworks, but also vice versa, to investigate if our concept of reversibility may improve the performance of existing tools (see how KLEE benefited from such a transfer of optimization ideas in [14]).

Acknowledgements: We thank Sorin Baltateanu and Traian Serbanuta for fruitful discussions and acknowledge partial support from MuVeT and MEASURE projects (PN-II-ID-PCE-2011-3-0688 and PN-III-P3-3.5-EUK-2016-0020).

References

1. European-Commission: Commission signs agreement with industry on cybersecurity and steps up efforts to tackle cyber-threats. `http://europa.eu/rapid/press-release_IP-16-2321_en.htm` (July 2016)
2. DARPA-US: Cyber grand challenge. `http://cgc.darpa.mil` (2016)
3. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2) (2013) 82–90
4. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4) (2009) 339–353
5. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. of ESEC/FSE, ACM (2005) 263–272
6. Cadar, C., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of OSDI, USENIX (2008) 209–224
7. Luckow, K.S., Pasareanu, C.S.: Symbolic PathFinder v7. ACM SIGSOFT Software Engineering Notes **39**(1) (2014) 1–5
8. Song, D., et al.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Proc. of ICCIS'08. Volume 5253 of LNCS., Springer (2008) 1–25
9. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on binary code. In: Proc. of SP'12, IEEE (2012) 380–394
10. Salwan, J., Saudel, F.: Triton: A dynamic symbolic execution framework. In: Proc. of, SSTIC (2015) 31–54 – Online at `http://triton.quarkslab.com`
11. Bitdefender: `http://www.bitdefender.com/business/awards.html` (2016)
12. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. Volume 4963 of LNCS., Springer (2008) 337–340
13. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. **30**(1) (2012) 2
14. Rizzi, E.F., et al.: On the techniques we create, the tools we build, and their misalignments: A study of KLEE. In: Proc. of ICSE'16, ACM (2016) 132–143
15. Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., Candea, G.: Cloud9: a software testing service. Operating Systems Review **43**(4) (2009) 5–10
16. Stephens, N., et al.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of NDSS'16, The Internet Society (2016) 1–16