Fundamenta Informaticae XX (2017) 1–19
DOI 10.3233/FI-2015-0000
IOS Press

Binary Analysis based on Symbolic Execution and Reversible x86 Instructions

Teodor Stoenescu
Bitdefender, Romania

Alin Stefanescu, Sorina Predut, and Florentin Ipate
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. In this paper we will provide details on the framework and give an example of analysis on binary code.

1. Introduction

Given the nowadays extreme interconnectivity between multiple systems, networks and (big) data pools, the field of cybersecurity is a vitally important aspect, in which concentrated efforts and resources are invested. As an example in this direction, the US recently organised, through DARPA, a cybersecurity grand challenge (CGC) [1], where successful teams competed to analyse and fix a benchmark of binary files using a combination of dynamic and static analysis, concolic execution, and fuzz testing.

Almost all the tools on the security market that 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 almost forty years ago [2, 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, 5]. 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. The main challenges of symbolic execution are the path explosion, constraint solving, and memory modeling. Symbolic execution is usually combined with taint analysis [6], which is a technique to determine the locations, i.e., memory and

registers, that have been directly influenced by certain values marked as tainted. (The basic algorithm for dynamic tainting works as follows: initially only certain locations are tainted; then, at runtime, any executed instruction with a tainted operand produces tainted results.)

Most of the symbolic execution tools work on source code or bytecode [7, 8, 9, 10, 11] rather than binary code [12, 13, 14, 15]. However, binary code analysis is a very difficult task due to its complexity and lower level constructs, so the challenges mentioned above are exacerbated. On the other hand, it is better to run the analysis directly at binary level, because this is the code that is executed by the operating system. Moreover, in the cybersecurity domain, in most cases, only the binary file is available for scrutiny, so recent research efforts are invested into dynamic analysis of binary files (see [1]) with some 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 [16]. Today, Bitdefender secures the digital experience of 500 million home and corporate users across the globe and, to keep its competitive edge, Bitdefender is constantly performing research and development activities in the software security area. The RIVER framework is an example of such internal research effort with 2.5 person-years invested in the project until now.

Contributions of the paper: From technical point of view, 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 this, the following concepts and components were created and implemented:

- the 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 (dynamic) taint analysis and symbolic execution engines based on the above representation;
- as a byproduct, a debugger at binary level with forward and backward step execution capabilities.

From scientific point of view, the main contribution of the paper is the investigation of the concept of reversible actions in the area of symbolic execution. Even though our implementation is for the x86 instruction set, we believe that this concept could be helpful for other symbolic execution engines, especially when the memory consumption becomes an issue. To better motivate this, we discuss at a very high level the advantages and disadvantages of some of the existing approaches. As mentioned above, the symbolic execution encounters problems due to path explosion or memory modelling, even when strategies such as selective path execution and path collapsing are applied [17]. One possible solution to these problems is to use concolic (i.e., concrete symbolic) execution. In this case, instead of being 100% symbolic, the inputs have a concrete value, which is a representative for a symbolic variable. The program is executed with concrete values, but constraints are accumulated along the path. Afterwards, some of these constraints are negated and a constraint solver is used to find new concrete input values for alternative paths. While fast on a single execution, the concolic approach suffers from a time execution problem because the system is restarted from the beginning with each concrete execution. One solution to this problem is to save snapshots of the two states after a branching occurs and restore them during the search. However, in this case a memory overhead problem may occur. As a possible solution to deal with this last problem, [13] introduced a hybrid approach, which alternates concolic execution with taking snapshots, when certain running time or memory thresholds are exceeded.

We propose a different solution to tackle the running time and memory overhead induced by the snapshots, using the concept of reversibility [18], i.e., recomputing a previous state, rather than saving it. Thus, there is no need to restart the executions from the beginning, nor to save snapshots (with only some exceptions). This offers an efficient implementation to local explorations using backtracking. However, the difficulty of this method is its engineering part, i.e., the requirement to implement specific inverse operations and dealing with other low-level details associated with that. We have done this for x86 instructions, but the reversibility principle is general and could be applied to other instructions sets such as ARM, although this will involve some extra engineering effort. Moreover, there are existing approaches investigating the usage of reversibility in x86 code, e.g., [19], but they are using it for debugging, not in combination with symbolic execution. This latter combination constitutes a novel scientific contribution of this paper.

The structure of the paper is the following. In the next section, we describe the RIVER framework. Then, a case study is presented in section 3 and related work in section 4, whereas the conclusions are reserved to the last section. Some technical details are provided in the Appendix. Note: This is the journal version of a short paper presented at Formal Methods 2016 (FM'16) conference, industrial track [20]. In that short version, only the overall RIVER architecture (section 2) is discussed, so the extended version has much more details (the size of the journal version is thrice the size of the short version).

2. Description of the framework

This section details the overall design of the RIVER framework, which is shown in Figure 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. The intermediate representation is depicted in the left hand side of Figure 1 and presented in subsection 2.1. It is obtained from an input as x86 native binary code (see bottom-left corner of Figure 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, described in subsections 2.2 and 2.3. All these are used by the symbolic execution engine which also uses a state-of-the-art SMT solver, Z3 [21], for dealing with the constraints for the symbolic variables (see top of Figure 1) but also on-demand snapshots to save certain memory states, as explained in subsection 2.4. Various other aspects and discussions about RIVER are provided in subsection 2.5.

2.1. RIVER intermediate language

Now we describe the RIVER intermediate language (IL) by presenting some design choices and a small example. First of all, RIVER code is obtained automatically from the input native x86 through the dynamic binary instrumentation component (DBI) which is plugged in the reversible execution component (see bottom-left corner of Figure 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 instructions. DBI also generates code for reversing the execution so that the destroyed values can be restored from the shadow stack.

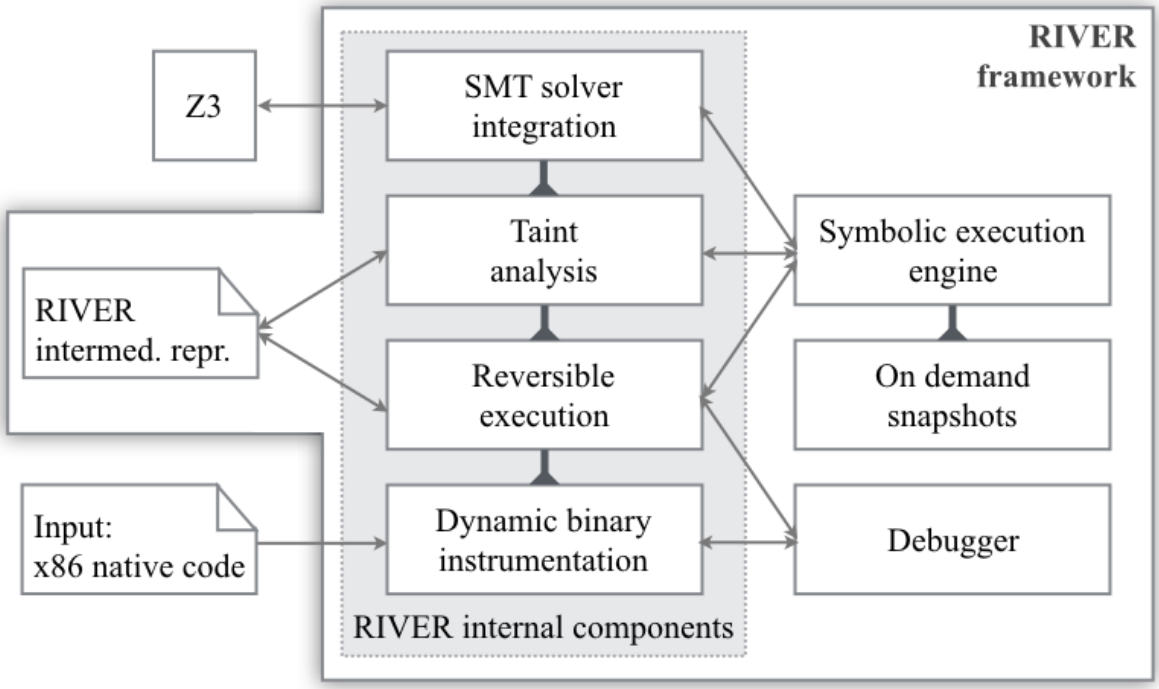

Figure 1. RIVER architecture

The RIVER instructions are fixed length instructions whose structure 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. The RIVER instructions are grouped in families, e.g., RIVER_FAMILY_NATIVE, RIVER_FAMILY_TRACK, or RIVER_FAMILY_RIVER. We will take a look at the last one, which enables reversible code.

The RIVER_FAMILY_RIVER family handles code reversibility by saving values that are about to be destroyed in an execution log. Reversing code execution simply translates to restoring the initial values. The available instructions are presented in Table 1. While reassembling, the RIVER instructions are converted into their native counterparts. The RIVER instruction set needs a separate ESP register. Transitioning to and from a RIVER instruction is achieved using a single `xchg` instruction. Table 2 shows two examples. The second example also shows how the register renaming works. For instance, non-native instructions make use of a modified stack pointer register (ESP). When these instructions need access to the original ESP, some form of register reallocation needs to be performed. The assembler makes use of one of the unused registers to hold the original ESP value.

RIVER dynamic binary instrumentation component also contains its own disassembler, which augments the code with the following properties:

- Implicit operands: some instructions implicitly modify registers and memory locations. These are added to the instruction as implicit operands.
- Register versioning: in order to simplify the data flow analysis, the disassembler versions every

Table 1. RIVER instructions dealing with code reversibility

Instruction mnemonic	Meaning
`riverpushf`	save the EFLAGS register
`riverpopf`	restore the EFLAGS register
`riverpush reg`	save a general purpose register
`riverpop reg`	restore a general purpose register
`riverpush mem`	save a memory location
`riverpop mem`	restore a memory location

Table 2. Two examples of reassembling

Original code	Reassembled code

`riverpushf [eax+4*ecx]`	`xchg esp, [espSave]`
	`push [eax+4*ecx]`
	`xchg esp, [espSave]`

`riverpush esp`	`xchg esp, [espSave]`
	`xchg eax, esp`
	`push eax`
	`xchg eax, esp`
	`xchg esp, [espSave]`

register use.

- Meta operations: since the x86 instruction set is not orthogonal, some instructions may be split into several sub-operations.
- 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.

Table 3 shows an example of decorated disassembled code from a simple basic block. On the output side (right side), an implicit operand appears on the second position of `push` (i.e., `esp$1, dword ptr [esp$0 + 0xfc]`), register versioning is given through the `$no` suffixes, meta operations are italic, and the inserted jump operand is bold. The next subsection shows how the reversibility is implemented.

2.2. Reversible execution component

The reversible execution engine (see middle of Figure 1) enables the forward and backward control of RIVER IL code which 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 (see the bold address at the bottom right of Table 3). RIVER also enables higher level components to augment and instrument code.

To implement reversibility, the RIVER translator inserts the RIVER instruction family (see Table 1) in the translated code (see the right side of Table 3). The output is presented in the left side of Table 4. The newly inserted instructions are bold. Then, the RIVER translator generates a second basic block for reversing the effects of the first block, which is shown on the right side of Table 4. Note also that the

Table 3. RIVER disassembling and decoration for a simple basic block

Original code	Disassembled decorated code
`mov edi, edi`	`mov edi$1, edi$0`
`push ebp`	*premetamov* `dword ptr[esp$0+0xfc], ebp$0`
`mov ebp, esp`	*premetasub* `esp$1, 4`
`cmp dword ptr[0x77b40150], 0x01`	`push ebp$0, {esp$1}, {dword ptr[esp$0+0xfc]}`
`jnz 0x000571d6`	`mov ebp$1, esp$1`
	`cmp dword ptr[0x77b40150], 0x01`
	`jnz 0x000571d6,` **0x77a79831**

Table 4. Forward and backward RIVER code for the basic block from Table 3

Forward RIVER code	Associated backward code
riverpush `edi$0`	`riverpopf`
`mov edi$1, edi$0`	`riverpop ebp$0`
`premetamov dword ptr[esp$0+0xfc] ...`	`espriverpop esp$0`
`premetasub esp$1, 4`	`espriverpop dword ptr[esp$0+0xfc]`
espriverpush `dword ptr[esp$0+0xfc]`	`riverpop edi$0`
espriverpush `esp$0`	`jmp 0x77a7981f`
`push ebp$0, {esp$1}, {dword ptr ...`	
riverpush `ebp$0`	
`mov ebp$1, esp$1`	
riverpushf	
`cmp dword ptr[0x77b40150], 0x01`	
`jnz 0x000571d6, 0x77a79831`	

translator operates only on native instructions and, as a general rule, every operand that is about to be overwritten has a corresponding `riverpush` instruction.

Based on the above, we developed a forward and backward binary debugger (see bottom-right of Figure 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. It allows the user to execute step by step through a binary, both forward and backward. 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. Thus, one can go back to a previous state and choose to explore another path from there.

Finally, we present some initial experiments that show the time savings through reversible execution when a whole test suite is executed. We have chosen two open source libraries, `http-parser`[1] and `libxml`[2] which were compiled to x86 binaries. Then, we generated automatically a test suite for each of them, using the fuzz testing tool `libFuzzer`[3]. The test suite for `http-parser` contained a corpus of 564 input data and the test suite for `libxml` contained a corpus of 2079 input data. The test suites were obtained by running `libFuzzer` overnight for the same period of time. Table 5 presents the running

[1] `https://github.com/nodejs/http-parser`
[2] `http://xmlsoft.org`
[3] `http://llvm.org/docs/LibFuzzer.html`

	baseline x86 code		RIVER code forward		RIVER code backward	
	http-parser	libxml	http-parser	libxml	http-parser	libxml
setup	134.49	499.51	162.13	595.90	0.29	0.28
meaningful	12.94	80.61	19.20	120.63	19.68	130.28

Table 5. Running times (in sec.) for the execution given test suites on two examples using three different modes

times, expressed in seconds, of executing the two libraries using the input data from their corresponding test suites. There are three modes in which we do this: first, we have the ”baseline x86 code” in which we just normally execute the library on the given inputs; then, we have ”RIVER code forward” which means we normally execute (i.e., forward) the RIVER code generated from the library code on the given inputs; and finally, for ”RIVER code backward”, we execute the library on each input data first forward and then backward. The table presents the cumulated running times for the test suites separately for the ”setup” and ”meaningful” phases. The ”setup” part is the time that has passed from the starting of the process until the ”main()” function (this includes creation of a new process, inserting of RIVER code for the last two modes and the initialisation of the process, including initialisation of native libraries or CRT initialisation). The so-called ”meaningful” part is the time that has passed from after ”main()” was executed. Now, we compare the times given in the table for the three modes. The time increase between the mode ”baseline x86 code” and ”RIVER code forward” reflects the time penalty induced by the extra RIVER code. On the one hand, the last mode, ”RIVER code backward”, shows an very small increase of the execution time, compared to ”RIVER code forward”, for the ”meaningful” part due to the fact that the RIVER code is executed not only forward, but also backward. On the other hand, the ”setup” part is extremely small, equating in fact the time of the ”setup” for only the first test, because the rest of the tests will not need to redo the setup, thus saving a lot of time. This experiment is an exemplification of the advantages of reverse execution over the concolic execution (mentioned in the introduction), in which the latter consumes a lot of time by starting always from the beginning.

2.3. Taint analysis component

This component records the spread of taint through a program which uses tainted values. We implement classic taint spreading algorithms [6], but we adapt them to our RIVER IL to take into account also the reversibility feature. Note that although there are many taint analysis systems, they are usually doing analysis on their intermediate representations [12, 8, 15], and few generic ones exist [22]. Since we do this at x86 assembly level, we have to deal with the complications that the instructions do not have simple semantics to know what registers are tainted by an operation (e.g., some instructions such as ”`xor eax, eax`” always produce the same results, so no taint propagation is performed here).

Technically, we also added tracking instructions in RIVER IL for the use of the taint analysis engine. DBI uses them in order to enable taint analysis, i.e., 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 (some exceptions might apply, see above). 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.

Without going into details because they are too technical, we mention that the tracking translator inserts instructions from a certain pretrack family in the original basic block and generates a separate block for tracking purposes. The tracking translator operates both on native and meta instructions. The main role of the pretrack instructions is to store a copy of every operand in a separate buffer (in order to save flag, registers or memory operands). These are later used in the tracking block where track instructions are generated on a per instruction basis. Each tracking block is split in two parts, the first part determines whether the original instruction has tracked operands and the second marks the output operands.

2.4. 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, 6, 3].

Our symbolic execution engine (see right of Figure 1) aims to tackle the path explosion problem through its distinctive feature of reversibility, introduced above. 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 optimize 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. As an example, suppose we have an input value controlled by the user. Some operations modify that value, which is ultimately compared against a constant. Afterwards a formula with the symbolic variable is bound to the input. This formula can be used to explore other execution paths.

It is a high priority for us to keep the snapshots at a minimum, and use it only on demand, 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 to compute and implement their inverse functions, wherever possible.

Regarding the symbolic execution engine, as most of current approaches, we do not implement "pure" symbolic execution, because it would be too expensive, but we 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. We use the taint analysis component to track the symbolic values. In this case the execution is sped up as the exercised code is run natively using concrete values, while taint analysis constructs are usually fast. RIVER constructs symbolic expressions based on the executed instructions. It keeps the symbolic side synchronized with the

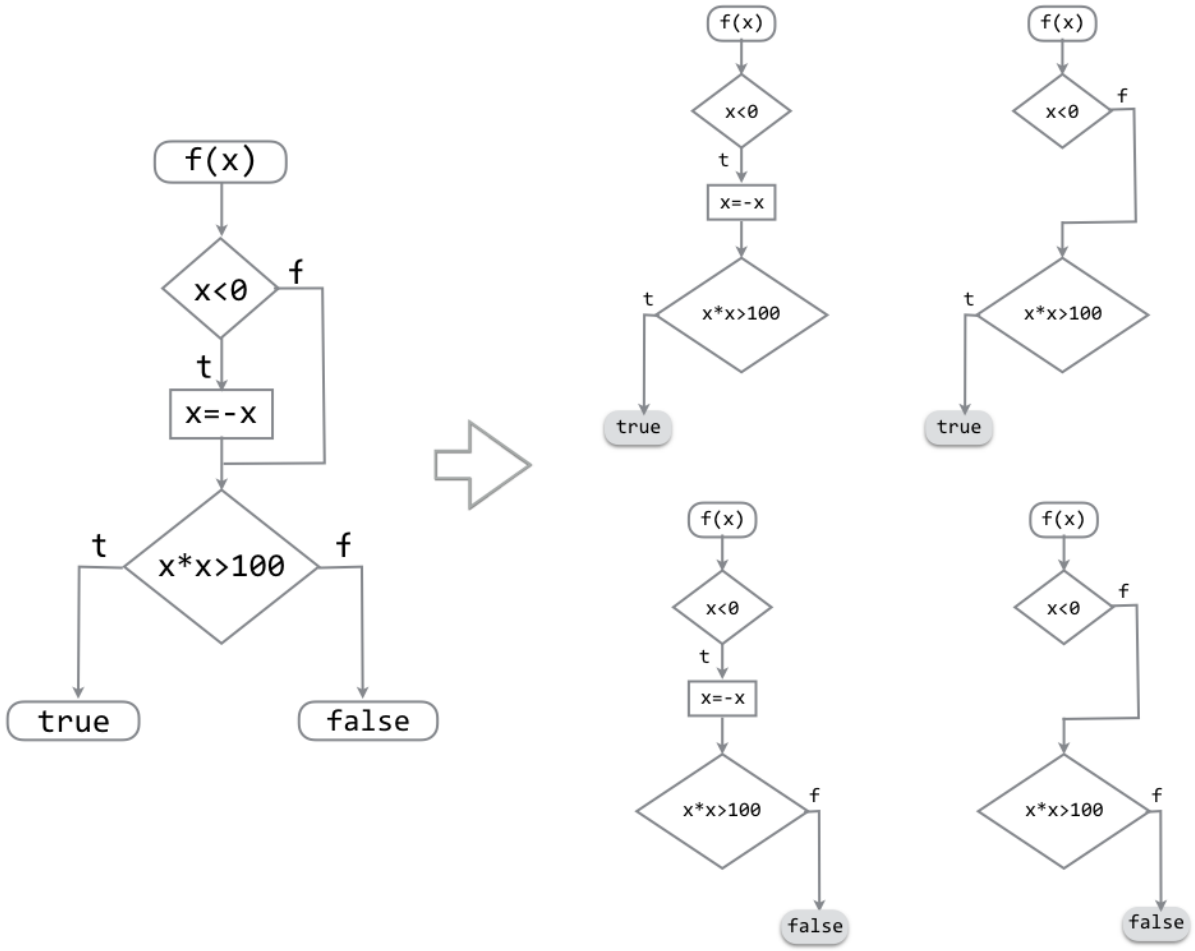

Figure 2. Control flow graph for a simple program with two conditions (left) and its four associated paths (right)

concrete one and uses one of the most capable SMT (satisfiability modulo theories) solvers available, Z3 [21] (see top of Figure 1).

Thus, the instrumented program runs concretely and also performs symbolic computation through the instrumented function calls. The symbolic execution follows the path taken by the concrete execution and replaces with the concrete value any symbolic expression that cannot be handled by the constraint solver. An instrumented program maintains two runtime states: the concrete one that can be directly executed and the symbolic one that maintains the current symbolic expressions.

We take the example program in Figure 2 to show the workings of concolic execution. For the program on the left, there are four execution paths possible, each with its distinct set of conditions. In a pure symbolic execution setting, the conditions collected along each path are fed into the SMT solver to determine whether they are satisfiable (and, if required, to generate the input test data). However, such approach has drawbacks. First, the number of paths can be extremely high and since the number of execution cores is much lower, some kind of scheduling is necessary and running time will be high.

Second, even if we parallelize a lot as in [23], duplicating the paths may be still accumulate and become costly. If not properly addressed, path explosion can be an issue so we have to use mitigation strategies like selective symbolic execution [24] which carefully picks execution paths to be evaluated and uses path collapsing, which finds common traits between paths and treat them as a single entity. A third issue is that there is no hardware support for symbolic values so the exercised code has to be run in a heavily simulated environment.

Getting back to our example, using concolic execution, the execution starts with a random value as input. Suppose that we have the assignment `X=5`. The program is started and conditions are accumulated at runtime. In this case, the expressions `!(X<0)` and `!(X*X>100)` are saved and at the end, the execution stops, also resulting a test case. Now, in order to discover other (different) test cases, an existing condition is selected, for instance the last one `!(X*X>100)`. Inverting the condition and using the SMT solver we obtain a new test case, if the accumulated conditions are satisfiable. In our case, the value provided by the SMT could be `X=12`. Again, the program is run with this value and conditions are accumulated at runtime, `!(X<0)` and `(X*X>100)`, resulting in a new test case. Now the other condition `!(X<0)` is selected for reversing and another concrete value (e.g., `X=-12`) returned by the SMT solver, and so on.

As you can see from above, we can reach with concolic execution the same test suite as in pure symbolic execution, without the need to simulate symbolic variables. The whole symbolic execution virtual machine is replaced with a symbolic context for accumulating conditions and with some method of keeping the symbolic context synchronised with the native execution.

2.5. 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. The architecture design permits easy developments on top of it, e.g., it is modular in the sense that internal components from middle of Figure 1 depend only on the components underneath.

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. To speed up implementation and increase this coverage, we explore also the integration with existing open source components for SMT integration, e.g., the ones from Triton [14].

Also, the symbolic execution engine implements only a straightforward depth-first exploration of paths using C APIs of the RIVER components in the middle of Figure 1, but we are now adapting several advanced features available in other state of the art symbolic execution frameworks [8, 3, 25, 1]. Note also that, beside the automated exploration, one can also manually select paths using the debugger component with its JavaScript bindings.

Moreover, we have now under development an integration of our concolic execution with parallel fuzz testing 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 test suite with a good coverage. Then, we apply symbolic execution by tweaking certain paths to increase coverage, as done also by others [23, 17]. But this is work in progress.

Now, RIVER IL increases between sixfold to ninefold the size of original x86 code. To lower this

```
unsigned int serial[] = {
    0x31, 0x3e, 0x3d, 0x26, 0x31
};

int Check(unsigned int *ptr) {
   int i, j = 0;
   int hash = 0xABCD;

   for (i = 0; ptr[i]; i++) {
      hash += ptr[i] ^ serial[j];

      j = (j == 4) ? 0 : j + 1;
   }
   return hash;
}
```

```
extern "C" unsigned int buffer[] = {
   'a','b','c','d','e','f','g','h',0
};

extern "C" int value = 0xad6d;

int Payload() {
   int ret;

   ret = Check(buffer);
   if (ret == value) {
      return 1;
   }
   return 0;
}
```

Figure 3. `Crackme` code

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. Regarding the runtime performance, we have some penalty proportional to the code size plus some extra overhead induced by the DBI. We are working on some heuristics involving caching to reduce this as well.

Note that we chose to lift RIVER instructions directly from x86 instructions and not from other IL such as LLVM or Vine IL from BitBlaze [12] in order to stay as close as possible to the x86 instruction set. However, this may have the disadvantage that porting RIVER to other instructions sets such as ARM may involve considerable engineering effort, but it would not be a problem from a conceptual point of view.

We do not use a whole-system emulator such as QEMU, because we work at the moment at application level. To reach whole-system level, we have to address the following aspects: multi-threading and multi-processing, synchronisation, and emulation of privileged x86 instructions which only run in kernel space.

3. A case study

In this section we show how RIVER works on a small example, using our current version of our symbolic execution engine which implements a depth-first strategy for generating the paths. We choose the `crackme` program used by others as well [15, 14], which (even though small) exemplifies some challenges in binary program analysis (more precisely, binary reverse-engineering challenges).

The program intends to find a string input which satisfies a certain property. The code in Figure 3 verifies the checksum of the user input against `0xad6d`. The `Payload()` function returns 1 if the input is good and 0 otherwise. One such good input is "`elite`", but there exist other solutions as well.

```
int main(int argc, char *argv[]) {

   InitializeRevtracer((rev::ADDR_TYPE)(&Payload));
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[0]), "a[0]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[1]), "a[1]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[2]), "a[2]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[3]), "a[3]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[4]), "a[4]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[5]), "a[5]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[6]), "a[6]");
   rev::MarkMemoryName((rev::ADDR_TYPE)(&buffer[7]), "a[7]");

   rev::Execute(argc, argv);
   return 0;
}
```

Figure 4. Marking of the variables as symbolic

In the Appendix, we give the details of the control flow graphs for the `Payload()` and `Check()` functions, with each rectangle box referring to a basic block.

Our concolic engine builds symbolic expressions on-the-fly when synchronizing the concrete and symbolic side. Z3 symbolic expressions are accessible through the use of opaque pointers. The taint analysis component binds these pointers to actual memory locations. Another reason Z3 has been chosen over similar SMT solvers is the fact that Z3 has a form of implicit garbage collection. Z3 has the concept of solver contexts, which support pushing and popping symbolic expressions, in a stack-like fashion. This enables us to keep the symbolic side synchronized with the concrete one, even when the execution is reversed. Reversing the symbolic side becomes just a matter of popping the solver context.

Figure 4 presents a snippet of the main function showing the C API usage for marking the variables as symbolic and starting the symbolic execution process.

While exploring a single execution path the conditions listed at the end of the Appendix have been accumulated. The first eight conditions represent prerequisites imposed by the nature of the program. All input values must either correspond to a lower case ASCII character or be NULL. The following conditions are accumulated while iterating over the input characters in the `Check()` function. The last condition is accumulated while comparing the output of `Check()` with the `value` variable inside the `Payload()` function.

We explored all execution paths in a depth-first manner. That means that we have fully exercised the `crackme` code obtaining a test case for each string length and for certain lengths we found strings matching the required value (`0xad6d`). The strings generated by RIVER are given in Table 6, with three solutions emphasized. Note that we could have stopped the engine after the first encountered solution, but we wanted to have the list of all test cases.

When a new test case is generated, the execution of the last basic block is reversed instead of restarting with a new path from the beginning, as mentioned in the previous section. Note that designing a good search heuristics is a major concern in dynamic symbolic execution [8, 25]. Many heuristics have been proposed in the literature such as best first, hybrid depth-first or generational search. In fact using only

Table 6. Strings generated by RIVER for `crackme`, with solutions emphasized in bold

`Check("abcdefgh")`	=	`0xae72`
`Check("hpkdmcc")`	=	`0xae17`
`Check("hpkdmc")`	=	`0xadba`
Check("qlyfys")	=	**0xad6d** ←
`Check("qlyfy")`	=	`0xad2b`
Check("jmyin")	=	**0xad6d** ←
`Check("jmyi")`	=	`0xad0e`
`Check("jez")`	=	`0xacca`
`Check("je")`	=	`0xac83`
`Check("j")`	=	`0xac28`
`Check("")`	=	`0xabcd`

one heuristic is not sufficient and depending on the program we should use more than one to achieve a better coverage. We will experiment within RIVER with such heuristics soon.

For our considered example, the total execution time was 136.58 milliseconds (ms) distributed as follows: the RIVER on-the-fly translation – 39.63 ms, the (concolic) execution of RIVER code – 0.32 ms, and execution of the SMT solver (i.e., Z3 running time) – 96.63 ms. These running times were obtained on a Intel Core i7 2600 @ 3.4GHz machine with 8 GB DDR3 RAM.

4. Related work

The dynamic symbolic execution research community was increasingly active over the last decade [7, 12, 8, 11, 4, 6, 5, 24, 13, 3, 14, 15], with the total number of published papers reaching soon two hundred. Since we cannot cover all the ideas recently developed, we refer the reader to the surveys [4, 3] and below we only present four frameworks, which are closest to RIVER.

BitBlaze [12] is a platform of binary analysis tools that combines different approaches of analysis. The platform is made up of three primary components for static analysis (Vine), the dynamic analysis component (TEMU), and the mixed concrete and symbolic analysis (Rudder). The system was developed to analyze x86 binaries and it has been used in academy and industry to find various types of bugs.

KLEE [8] is a symbolic execution platform that was build up to perform deep checking of applications and maximize code coverage across various classes of programs written in C. A program under test is compiled into LLVM bytecode, then KLEE performs the symbolic execution using a symbolic virtual machine capable of running LLVM code together with SMT solver for the path constraints. Since KLEE is based on LLVM IL, we cannot apply it directly to RIVER IL. However, we plan to integrate or adapt some of its strong optimization strategies [25].

S2E [24] is a platform for writing tools that analyze the properties and behavior of software systems. S2E is a virtual machine augmented with symbolic execution and modular path analyzers. S2E runs unmodified x86, x86-64, or ARM software stacks, including programs, libraries, the kernel, and drivers. This is possible through a x86-to-LLVM translator and a modified QEMU for running a full operating

system. Symbolic execution, based on KLEE, then automatically explores large number of paths through the system, while analyzers check that the desired properties hold on these paths and selectors focus path exploration on components of interest.

Triton [14] is a new open-source Pin-based concolic execution framework, sponsored by security company Quarkslabs. Triton has similar components to RIVER internal components, translation of x64 instruction into SMT2-LIB, a Z3 interface to solve constraints. Triton provides extension points and Python bindings which enables others to build on top of it tools for automated reverse engineering or vulnerability analysis. We are investigating if we can plugin in RIVER Z3 integration component

MAYHEM [13] is a recent tool for binary analysis developed in academia and under continuous improvement at a spin-off company from the Carnegie Mellon University, called ForAllSecure. MAYHEM uses a hybrid symbolic execution approach, by alternating between online and offline symbolic execution runs. It is based on the following four principles: (a) make continuous progress without exceeding the given resources (especially memory), (b) no work should be repeated to increase performance, (c) previous analysis results of the system should be reusable on subsequent runs, and (d) the system should be able to reason about symbolic memory, especially for the difficult case where user input is involved. Interestingly, they are very similar to RIVER design choices, although RIVER is based on the concept of reversible executions.

The most used approach of current frameworks [6] is to keep different states or snapshots (depending on the implementation) when traversing the execution tree using different strategies such as depth-first, breadth-first or random paths. As opposed to that, RIVER always retains only one path and uses backtracking and reversibility to generate new paths and associated states. Thus, we save space by keeping the footprint of reverse operation minimal and we save running time by generating code dedicated to each basic block for restoring the state. Also, we invested actively in reducing the occurrence and size of the snapshots, and not rely on expensive snapshots. However, we cannot yet compare against the above tools because our symbolic execution engine is still under development and our SMT integration does not have a high coverage.

Finally, we mention that the idea of reversible computation is not new [18], with a dedicated conference series reaching its 9th edition [26], but we have not found any work trying to combine reverse computation with symbolic execution as we do in the RIVER framework. The closest, but still different, approaches were the application of reverse execution for debugging [19], or specification of dynamically generated reverse code [27], or using constraint solving to improve reverse execution for debugging [28]. Note also that our approach of reverse execution for symbolic execution is different from the so-called ”backward symbolic execution” [29, 30] in the sense that our approach can be executed backwards only after it was executed forwards using an ”undo” mechanism, whereas backward symbolic execution does not need forward execution (albeit it may suffer from the similar problems as forward execution).

5. 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. All 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 which were already mentioned throughout the sections 2.5 and 4 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 in any way the performance of existing tools (see how KLEE benefited from such a transfer of optimization ideas in [25]).

Acknowledgements We warmly thank our colleagues Sorin Baltateanu, Traian Serbanuta, and Alexandra Sandulescu, with whom we discussed about various aspects of the framework and students at the University of Bucharest for investigating certain technical aspects of the approach. We also acknowledge the partial support of research grants: MuVeT (PN-II-ID-PCE-2011-3-0688) and MEASURE (PN-III-P3-3.5-EUK-2016-0020).

References

[1] DARPA US. Cyber Grand Challenge; 2016. `http://cgc.darpa.mil`.

[2] King JC. Symbolic Execution and Program Testing. Commun ACM. 1976;19(7):385–394.

[3] Cadar C, Sen K. Symbolic execution for software testing: three decades later. Commun ACM. 2013;56(2):82–90.

[4] Pasareanu CS, Visser W. A survey of new trends in symbolic execution for software testing and analysis. STTT. 2009;11(4):339–353.

[5] Cadar C, Godefroid P, Khurshid S, Pasareanu CS, Sen K, Tillmann N, et al. Symbolic execution for software testing in practice: preliminary assessment. In: Proc. of ICSE'11. ACM; 2011. p. 1066–1071.

[6] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. of SP'10. IEEE; 2010. p. 317–331.

[7] Sen K, Marinov D, Agha G. CUTE: a concolic unit testing engine for C. In: Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM; 2005. p. 263–272.

[8] Cadar C, Dunbar D, Engler DR. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of OSDI'08. USENIX Association; 2008. p. 209–224.

[9] Tillmann N, De Halleux J. Pex: White Box Test Generation for .NET. In: Proc. of TAP'08. vol. 4966 of LNCS. Springer; 2008. p. 134–153.

[10] Li G, Ghosh I, Rajan SP. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In: Proc. of CAV'11. vol. 6806 of LNCS. Springer; 2011. p. 609–615.

[11] Luckow KS, Pasareanu CS. Symbolic PathFinder v7. ACM SIGSOFT Software Engineering Notes. 2014;39(1):1–5.

[12] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, et al. BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Proc. of ICCIS'08. vol. 5253 of LNCS. Springer; 2008. p. 1–25.

[13] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing Mayhem on Binary Code. In: Proc. of SP'12. IEEE; 2012. p. 380–394.

[14] Salwan J, Saudel F. Triton: A Dynamic Symbolic Execution Framework. In: Symposium sur la sécurité des technologies de l'information et des communications. SSTIC; 2015. p. 31–54 – Online at `http://triton.quarkslab.com`.

[15] David R, Bardin S, Ta TD, Mounier L, Feist J, Potet M, et al. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In: Proc. of SANER'16. IEEE; 2016. p. 653–656.

[16] Bitdefender. Top awards received by Bitdefender from independent evaluators; 2016. `http://www.bitdefender.com/business/awards.html`.

[17] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: Proc. of NDSS'16. The Internet Society; 2016. p. 1–16.

[18] Bennett CH. Time/space trade-offs for reversible computation. Siam Journal on Computing. 1989;18:766–776.

[19] Akgul T, III VJM. Assembly instruction level reverse execution for debugging. ACM Transactions on Software Engineering and Methodology. 2004;13:149–198.

[20] Stoenescu T, Stefanescu A, Predut S, Ipate F. RIVER: A Binary Analysis Framework Using Symbolic Execution and Reversible x86 Instructions. In: Proc. of FM'16. vol. 9995 of LNCS. Springer; 2016. p. 779–785.

[21] de Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Proc. of TACAS'08. vol. 4963 of LNCS. Springer; 2008. p. 337–340.

[22] Clause JA, Li W, Orso A. Dytan: a generic dynamic taint analysis framework. In: Proc. of ISSTA'07. ACM; 2007. p. 196–206.

[23] Ciortea L, Zamfir C, Bucur S, Chipounov V, Candea G. Cloud9: a software testing service. Operating Systems Review. 2009;43(4):5–10.

[24] Chipounov V, Kuznetsov V, Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Trans Comput Syst. 2012;30(1):2.

[25] Rizzi EF, Elbaum S, Dwyer MB. On the techniques we create, the tools we build, and their misalignments: A study of KLEE. In: Proc. of ICSE'16. ACM; 2016. p. 132–143.

[26] 9th Reversible Computation Conf.; 2017. Available from: `http://www.reversible-computation.org`.

[27] Lee J. Dynamic Reverse Code Generation for Backward Execution. Electr Notes Theor Comput Sci. 2007;174(4):37–54.

[28] Sauciuc R, Necula G. Reverse Execution With Constraint Solving. University of California at Berkeley; 2011. Tech report no. UCB/EECS-2011-67.

[29] Holzmann GJ. Backward Symbolic Execution of Protocols. In: Proc. of the IFIP WG6.1 Fourth International Workshop on Protocol Specification, Testing and Verification. North-Holland; 1985. p. 19–30.

[30] Gulwani S, Juvekar S. Bound Analysis using Backward Symbolic Execution. Microsoft Research; 2009. Techreport no. MSR-TR-2009-156.

Appendix

In this Appendix we provide lower level details complementing Subsection 3.

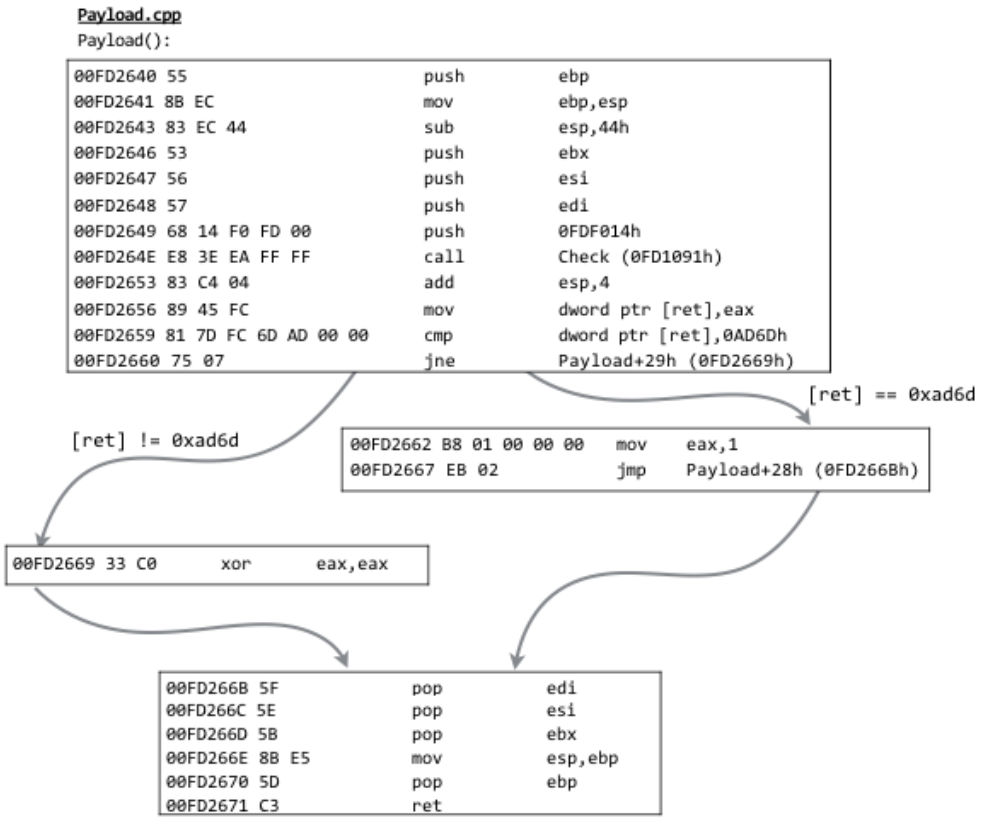

Figure 5. `Payload()` assembly code

We start by giving in Figure 5 the details for the control flow graph of the `Payload()` function and in Figure 6 those of `Check()` function, with each rectangle box referring to a basic block. The `Payload()` function starts at location `0x00FD2640`. The first basic block contains instructions and among them the `Check()` function is called from `0x00FD264E` and the verification of its result against `0xad6d` is made at `0x00FD2659` in the conditional statement. If the returned value of `Check()` is `0xad6d` then we take the branch which starts at `0x00FD2662` and return 1 then jump to the next block at `0x00FD266B`. On the contrary, if the value is not equal to `0xad6d` we take the other branch whose address is `0x00FD2669` and the instruction in that block means the end of the function (the returned value is 0; `xor` between two equal values is 0).

Next, we discuss the control flow graph of the `Check()` function called in `Payload` function, see Figure 6. In the basic block starting at `0x00FD25A0` we have instructions in which the variables `j` and `i` are initialized with 0 (`0x00FD25A9`, `0x00FD25B7`) and `hash` is assigned the `0xABCD` value at `0x00FD25B0`. From this block we jump to another where there is a conditional statement for the parameter of the function. If the value at `0x00FD25CF` is not equal to zero, then we jump to the block whose initial location

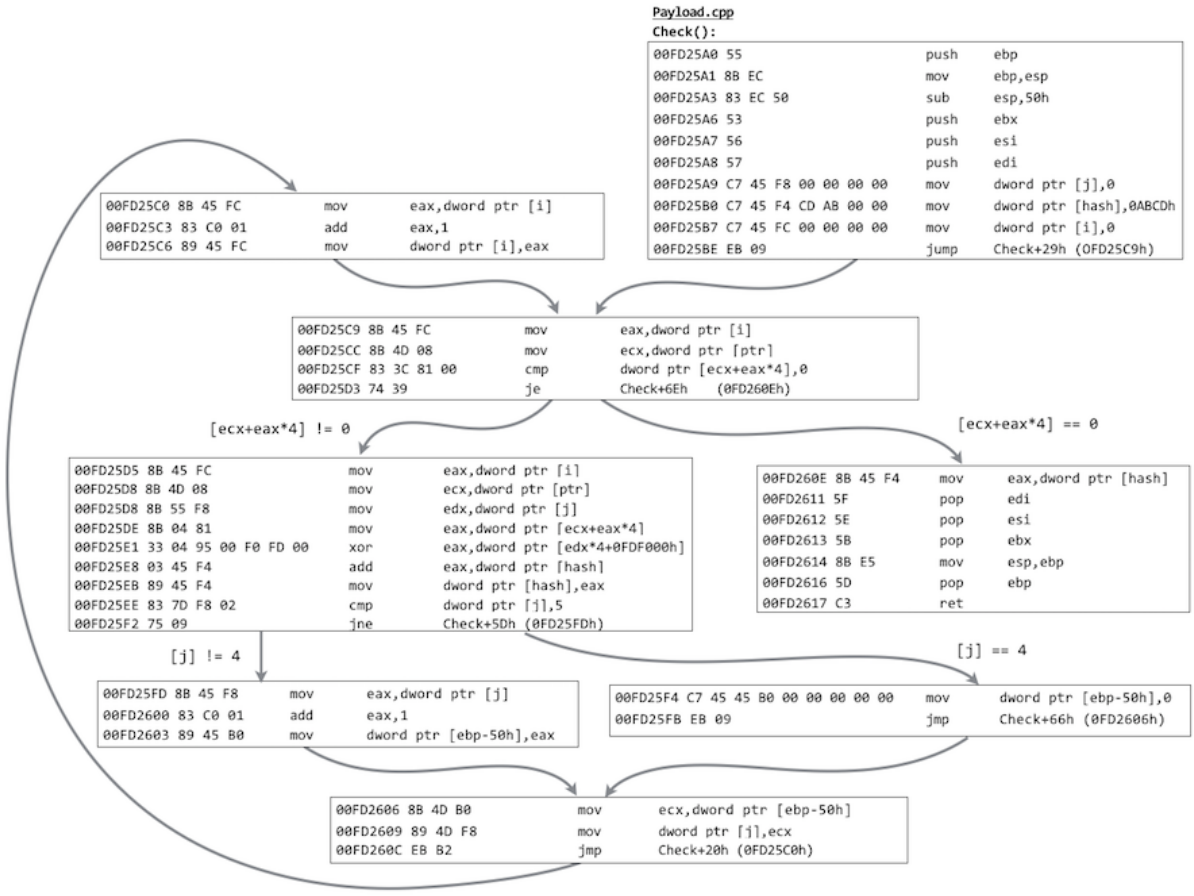

Figure 6. `Check()` assembly code

is `0x00FD2505`. There are two important statements executed there. At `0x00FD25E1` every character of the formal parameter `ptr`, which is actually the external variable `buffer`, is xored with a string `serial` which is equivalent to the ASCII string "`1>=&1`". At `0x00FD25EE` there is a conditional statement for the value of `j` variable. If it is 4, then we jump to the block starting at `0x00FD25F4` because the aforementioned string has 5 characters and so we have to set 5 to 0 in order to obtain the remainder of the division by 5. Otherwise, i.e., `j` is not equal to 4, then we increment it in the block starting at `0x00FD25FD`. Both cases lead to the block whose initial location is `0x00FD2606` where we update the value of the character of the string corresponding to `j` index. We jump to the block whose location is at `0x00FD25C0` where we increment the `i` index and start looping again until the loop condition is false (the process was repeated until every character of `buffer` was processed) returning the value `hash` which is done in the block with initial location `0x00FD260E`. `Hash` is the sum of characters of the parameter provided by the `Check()` function.

```
(assert (or (= #x00000000 |a[0]|)
    (and (bvule #x00000061 |a[0]|) (bvuge #x0000007a |a[0]|))))
(assert (or (= #x00000000 |a[1]|)
    (and (bvule #x00000061 |a[1]|) (bvuge #x0000007a |a[1]|))))
... snip ...
(assert (or (= #x00000000 |a[7]|)
    (and (bvule #x00000061 |a[7]|) (bvuge #x0000007a |a[7]|))))

(assert (not (= |a[0]| #x00000000)))
(assert (not (= |a[1]| #x00000000)))
... snip ...
(assert (not (= |a[7]| #x00000000)))

(assert (let ((a!1 (bvadd (concat ((_ extract 31 6) |a[7]|)
                          (bvnot ((_ extract 5 1) |a[7]|))
                          ((_ extract 0 0) |a[7]|))
                  (concat ((_ extract 31 6) |a[6]|)
                          (bvnot ((_ extract 5 4) |a[6]|))
                          ((_ extract 3 1) |a[6]|)
                          (bvnot ((_ extract 0 0) |a[6]|)))
                  (bvxor |a[5]| |a[0]|)
                  (concat ((_ extract 31 6) |a[4]|)
                          (bvnot ((_ extract 5 4) |a[4]|))
                          ((_ extract 3 1) |a[4]|)
                          (bvnot ((_ extract 0 0) |a[4]|)))
                  (concat ((_ extract 31 6) |a[3]|)
                          (bvnot ((_ extract 5 5) |a[3]|))
                          ((_ extract 4 3) |a[3]|)
                          (bvnot ((_ extract 2 1) |a[3]|))
                          ((_ extract 0 0) |a[3]|))
                  (concat ((_ extract 31 6) |a[2]|)
                          (bvnot ((_ extract 5 2) |a[2]|))
                          ((_ extract 1 1) |a[2]|)
                          (bvnot ((_ extract 0 0) |a[2]|)))
                  (concat ((_ extract 31 6) |a[1]|)
                          (bvnot ((_ extract 5 1) |a[1]|))
                          ((_ extract 0 0) |a[1]|))
                  (concat ((_ extract 31 6) |a[0]|)
                          (bvnot ((_ extract 5 4) |a[0]|))
                          ((_ extract 3 1) |a[0]|)
                          (bvnot ((_ extract 0 0) |a[0]|)))))))
  (not (= a!1 #x000001a0))))
```