rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery

Alessandro Di Federico[§†], Pietro Fezzardi[§†], Giovanni Agosta[§]

[§]DEIB – Politecnico di Milano, [†]rev.ng Srls Unipersonale.
email: name.surname@polimi.it

August 1, 2018

Abstract—**Reverse engineering is a key technique to perform security audits and malware analysis. However, existing tools have severe limitations in terms of recovering a correct, semantics-preserving representation of the program behavior.**

This work introduces rev.ng [1] [2], a reverse engineering framework based on QEMU [3] and LLVM [4]. QEMU is an emulator able to handle about 20 distinct architectures. We employ it to translate machine code into an Intermediate Representation (IR) independent from the input architecture. Then, we transform this representation into the IR employed by LLVM, a robust open source compiler framework. This allows rev.ng to easily recompile the obtained representation to any of the 10 architectures supported by LLVM.

Currently, we can analyze large programs (e.g., GCC and Perl) and recompile them, obtaining programs that preserve their original behavior. This allows us to quickly validate the analyses that we designed. In fact, one of the key goals of rev.ng is to preserve the semantics of the analyzed program across all the transformations performed on the code.

As a consequence, we can instrument programs to get deeper insights on their behavior. In addition, we can identify functions, their inputs, and try to generate input values that lead the program into an invalid state. Such inputs can be generated through coverage-guided fuzzing, a technique known for being extremely effective in finding security vulnerabilities.

In this paper we introduce the architecture of rev.ng, demonstrate the performances of the generated code, and discuss a case study, to show how we can easily generate inputs triggering invalid program states using the above mentioned techniques.

1. Introduction

Nowadays, many reverse engineering tools are available on the market. Such tools provide extremely useful insights for the analysis of computer programs whose source code is not available, enabling the understanding of their structure and behavior. The most useful among such tools are *decompilers*, i.e., tools that recover a representation of the program's code in a programming language easier to understand compared to raw assembly, typically C.

One of the key goals of such tools is to provide abstraction as high level as possible (e.g., control-flow constructs such as `for` loops) to the analysts. However, to achieve this goal, reverse engineering tools often push their assumptions too far. As a result, the resulting decompiled code doesn't preserve the semantics of the original binary program, or, in many cases, it is not even suitable for being recompiled. In fact, this is the case for IDA Pro [5], the de-facto standard and industry leader in the field of decompilers.

Unlike such tools, rev.ng is a reverse engineering framework which aims at preserving the semantics of a program across all of its analyses. This is possible thanks to the use of the LLVM IR as our internal representation. The LLVM IR is the representation used by the LLVM compiler framework [4], which is the foundation for many mature and robust compilers, first and foremost the C/C++ clang compiler. Since the whole analysis pipeline in rev.ng works on such representation, it is extremely easy to employ LLVM to recompile the code. In this way, we can generate a new binary executable, possibly for a different architecture, and verify that its behavior has not changed.

Thanks to this approach we can, not only continuously check the correctness of our analyses, but also integrate a set of features that are not traditionally available in reverse engineering tools but only in tools for automatic bug detection. A very relevant approach for this kind of tasks is *coverage-guided fuzzing*, which is already maturely supported in LLVM thanks to libFuzzer [6].
The main contribution of this work are:

- the introduction of the rev.ng unified binary analysis and translation framework (Section 2);
- the extension of rev.ng's capabilities with automated coverage-guided fuzzing to target dedicated functions in the program (Section 3);
- the evaluation of the run-time performances of the programs translated with rev.ng compared to traditional emulation, reaching an average speedup of $1.65\times$ on average, with a maximum of $2.88\times$ (Section 4.1);
- the evaluation of rev.ng's coverage-guided fuzzing extension on a real-world vulnerability (Section 4.2), compared to other state-of-the-art fuzzing techniques on binaries.

2. rev.ng Architecture

The core component of rev.ng is the *static binary translator*, a tool that, given an input program for a certain architecture, produces a program with an equivalent behavior for another architecture (or even the same). Figure 1 offers an overview of its translation process.

The translator takes a statically linked GNU/Linux program and identifies the portions of the program containing

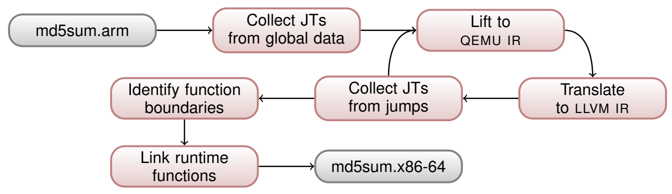

Figure 1. Overview of the static binary translation process.

data and containing code. A pre-processing phase collects an initial set of *jump targets* from the global data and the program's entry points. *Jump Targets* (JT in Figure 1) are the addresses in the binary where the execution can jump to, i.e., the start addresses of basic blocks.

Then, an iterative discovery of *jump targets* begins. The code at the address corresponding to each jump target is first translated into *tiny code instructions*, i.e., the IR used in QEMU to represent instructions in a format that is independent of the architecture. This allows us to start the process from virtually any architecture supported by QEMU: x86, x86-64, ARM, AArch64, s390x, MIPS, and many others. In a second step, QEMU's *tiny code instructions* obtained from the binary are translated into LLVM IR. The LLVM IR is collected in a single, large, recompilable LLVM module. The module is then further analyzed to recover additional *jump targets*. When no new *jump targets* can be found, further analyses are performed to identify the function boundaries and other high-level constructs. Finally, the program is linked against the necessary runtime libraries which implement system calls and complex instructions (e.g., floating point division) and the final executable is emitted, targeting one of the many architectures supported by the LLVM compiler framework.

The end result is a new program, possibly targeting a different architecture, whose behavior closely mimics the original program.

3. Non-static analysis applications

While `rev.ng` is primarily a tool for static analysis, it is not limited to it. In particular, in this paper we show how it can be employed to identify security-critical vulnerabilities.

To perform this task, it is interesting to devise inputs that lead the program to an invalid state, such as overwriting part of a buffer located on the stack or on the heap, access invalid memory areas and so on. Such situations constitute the starting point of the bug hunting process, since they are often a sign of an exploitable memory vulnerability. One of the most popular and effective techniques to artificially trigger invalid program states is *fuzzing*.

Fuzzing. *Fuzzing* has recently gained very high traction [7] due to large amount of vulnerabilities it enabled to discover, thanks to a relatively new approach known as *coverage-guided fuzzing*. Coverage-guided fuzzing basically consists in automatically generating a large amount of inputs by mutating valid inputs, and privileging inputs that lead program execution towards previously unexplored portions of the program. This increase in the coverage of the program can be detected only by instrumenting the target program as appropriate.

The most notable coverage-guided fuzzers are `afl` (american fuzzy lop [8]) and `libFuzzer` [6]. They are both well-known for finding several bugs in many high-profile open source projects but, except for being both coverage-guided fuzzers based on instrumentations, they are quite different.

`afl`, provides two distinct modes of operations: the former mode is based on a compiler plugin which injects instrumentations to collect coverage during compilation; the latter mode, known as *black-box mode* or *QEMU-mode*, directly instruments the binary and can work without the availability of the program's source code, using QEMU [3] as an emulator. The QEMU approach is more generally applicable, but incurs in a non-negligible slowdown, effectively reducing the effectiveness of the fuzzer. Both `afl`'s modes can only work fuzzing the entry point of the program, and they provide input vectors through files or standard input.

On the other hand, `libFuzzer`, being based on LLVM, requires the source code to inject the instrumentation during the compilation process. However, this limitation is compensated by a significant advantage: `libFuzzer` can be used to fuzz user-defined entry points in a program, providing input vectors by means of function arguments, which results in a considerable speedup compared to `afl`. Users are only required to write shallow wrappers around the selected entry point in C/C++, and they can directly start fuzzing even functions whose calls would otherwise be deeply nested in the program logic.

This approach have shown great results in finding vulnerabilities, but applying it end-to-end on a full program can be difficult and might require a prohibitive amount of resources. This might happen due to costs concerning the program setup or complex and time-consuming computations on the inputs before an interesting portion of the program is reached. In some cases, such as firmware of embedded systems, it might not even be possible to run a program end-to-end.

It is therefore sometimes useful, while performing reverse engineering activities on unknown programs, to perform fuzzing on individual functions, instead of the program as a whole.

3.1. `rev.ng` and `libFuzzer`

The `rev.ng` workflow allows to take one of the functions identified by the tool, choose some of its arguments, and fuzz it as if they were buffers under direct control of an attacker.

This places `rev.ng` in a unique position in the field of binary analysis tools, taking both the advantages of `afl` and `libFuzzer`. In fact, it offers the possibility to fuzz binaries for which the source code is unavailable, starting from user-defined entry points. This is particularly beneficial when, after a reverse-engineering phase, the analyst identifies a function with an argument (a value or a buffer) that is under his/her full control. In this case, the analysis typically wants to determine if it is possible, by crafting the inputs, to reach invalid program states that can lead to vulnerabilities.

More specifically, the fuzzing workflow employing rev.ng and libFuzzer is as follows:

Translation. The input program is fed to rev.ng which produces an LLVM IR module that can be easily recompiled into a working program.

Entry point identification. A manual reverse engineering phase is performed to identify functions that might be interesting to fuzz.

Creation of the fuzzing function. libFuzzer requires a single entry point named LLVMFuzzerTestOneInput which takes two inputs: a pointer to a buffer and the corresponding size. This function will be invoked multiple times by the libFuzzer runtime with different inputs, generated by mutation and privileging those leading to an increase of program coverage. It's up to the user to implement this function. Basically, it should prepare the environment as appropriate and then invoke the target functions using the given input.

Compilation of the fuzzer. The fuzzing function is linked against the LLVM IR module produced by rev.ng.

Fuzzing. The resulting program is executed until it crashes.

Usually libFuzzer requires the source code of the program to inject the instrumentations necessary to detect which parts of the program have been executed (covered) due to a certain input. However, the libFuzzer instrumentation pass manipulates the LLVM IR. Therefore, thanks to the fact that rev.ng internally uses the LLVM IR, we are able to extend rev.ng to activate libFuzzer even without the availability of the source code.

Note that the LLVMFuzzerTestOneInput has to call functions within the module produced by rev.ng. To do this each function call has to be set up by a piece of code manually preparing arguments in the appropriate registers and then calling the corresponding code in the translated program, effectively running the target function. At this stage, the argument set-up phase is performed manually by writing the argument values in the corresponding registers, however, in the future, thanks to our in-progress argument detection analysis, this won't be necessary and the function invocation will look like a normal function call.

Note also that, before testing the first input, we also perform an initialization step which basically lets the translated program run until the main function is met. This allows to correctly initialize the C standard library of the target program (which is distinct from the one employed by the fuzzer) and, specifically, the memory allocator, which would otherwise be unusable. This is the case since we statically linked our target. It is possible to use dynamically linked programs and hook library functions in the appropriate way, making the fuzzing process faster. However, this approach would be less general.

4. Experimental Evaluation

In this section we evaluate the proposed approach. Section 4.1 discusses the performance, compared to QEMU (which is employed by afl in black-box mode) which operates on binaries without source code like rev.ng. Section 4.2 is a report on a case study, where rev.ng is used to independently find a CVE, showing the benefits that its workflow brings to the analysis of software whose source code is not available.

All the measurements were gathered on a GNU/Linux system with Ubuntu 16.04 LTS on an Intel Xeon CPU E3-1220 v6 running at 3.00 GHz, with 32 GB of DDR4 RAM (2133 MHz) and a 2 TB HDD (5400 RPM) attached to SATA 6 Gb/s. All the benchmarks have been executed on a single core, being the only active workload on the test machine. LLVM 3.8 has been employed both for rev.ng and libFuzzer.

4.1. Performance

One of the key factors affecting the effectiveness of a fuzzer is its performance. In fact, the instrumentations used to collect coverage data to guide fuzzing impose an overhead. The number of inputs that can be tested in a certain amount of time is fundamentally a metric of how fast a fuzzer can explore new possible executions. Therefore, it makes sense to compare the performance of the code produced by rev.ng against the performance of the original code and the performance of QEMU, which is employed by afl in the black box mode.

Figure 2 reports the comparison of the performance obtained on the SPEC2006 [9] integer benchmark suite. The picture shows the runtime obtained by native code, code emulated with QEMU and code translated with rev.ng. The results are collected over three runs for each benchmark, and aggregated using the geometric mean.

The code generated by rev.ng is generally sensibly faster. Specifically, the average overhead introduced by QEMU is $5.680\times$ with respect to native code, while the average slowdown introduced by rev.ng is only $3.439\times$. In other words, code translated by rev.ng runs on average $1.65\times$ faster than code emulated with QEMU. In the best scenario, (400.perlbench) rev.ng provides a speedup of $2.88\times$ compared to QEMU. This is in part due to the optimizations that LLVM can perform thanks its view over the whole program code, as opposed to QEMU which works on a basic block granularity, and in part due to the fact that rev.ng performs the whole translation offline, paying the associated overhead only once at translation-time, and not during the execution.

4.2. Case study

To demonstrate the effectiveness of the targeted fuzzing approach adopted in rev.ng (see Section 3), and the boost it can give to program analysis tasks, we used it to analyze a stack-based buffer overflow in the Perl Compatible Regular Expressions library known as PCRE [10] (CVE-2015-3217 [11]). This library is ubiquitous and used in many high-profile projects, such as PHP, the Apache HTTP server, Apple's browser Safari, and others.

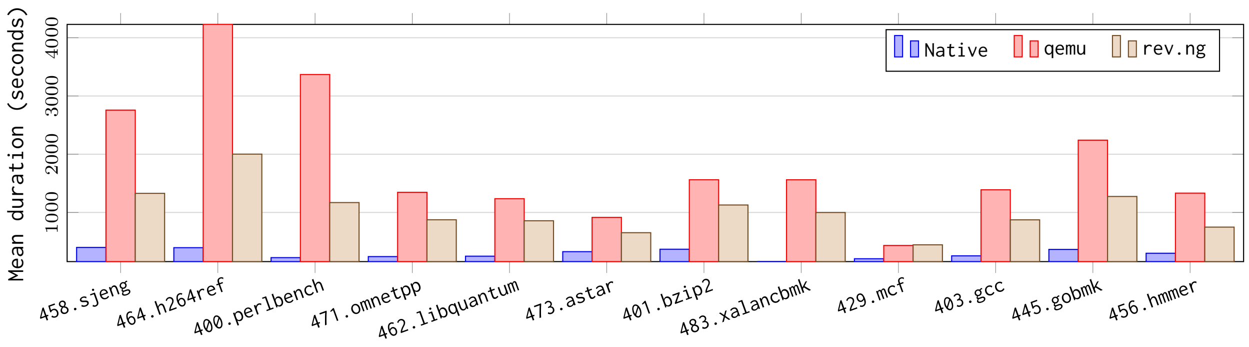

Figure 2. Comparison of the performance of the integer SPEC2006 benchmarks over three runs. 🟦 represents the performance of the native x86-64 program, 🟥 the performance of the same program running under QEMU and 🟫 the performance of the program translated by `rev.ng`.

Our experiments aim at showing mainly three things: 1) by using `rev.ng` with targeted coverage-guided fuzzing based on `libFuzzer` [6] it is possible to independently find the bug reported in the CVE; 2) it is possible to identify the bug even without access to the source code; 3) the presented approach lowers the effort required to an analyst to set up fuzzing for reverse engineering tasks on binaries without source code, compared to manually instrumenting the binary or using `afl`'s QEMU-mode.

The `rev.ng` and `libFuzzer` approach. For our analysis we took the PCRE library version 8.37 [10], which was known to be affected by the vulnerability [11]. We then took the latest version of the `less` command line file visualization tool, which supports text search using PCRE. Then, we built `less` using the flawed library, compiling all the sources as a static executable for GNU/Linux on x86-64. In this way, for the analysis of the `less` program we had put ourselves in the same conditions of an analyst looking for the first time at an entirely new program. Initially, we used `rev.ng` on the binary to translate it. The idea is to now identify within the binary the relevant `libcpre` entry points in the LLVM IR module, to then employ them as fuzzing targets. This could be easily achieved with a moderate reverse engineering effort, but in our case, we simply employed debug information to identify them.

Once the entry-point for fuzzing has been found, the analyst has to write the `LLVMFuzzerTestOneInput` wrapper that will be the entry-point for `libFuzzer`. For our example, the three core functions we want to stimulate are `pcre_compile`, which, given a string, *compiles* a regular expression object and returns it, `pcre_exec`, which runs a previously compiled regular expression on a string and finally `pcre_free`, which releases the memory associated with the compiled regular expression. The invocation mechanism is quite simple and is illustrated in Listing 1.

Note that invoking `pcre_free` is not optional. In fact, unlike when fuzzing with `afl`, where a new process is spawned for each new input (leaving it to the operating system to cleanup the allocated resources), `libFuzzer` tests many inputs in the same process. Therefore, if we didn't take care of freeing the resources allocated by each run, we would quickly run out of memory. This approach, despite making the fuzzing process more *fragile*, can greatly

```c
int LLVMFuzzerTestOneInput(uint8_t *data,
                           size_t size) {
  if (size > 1024) return 0;

  char re[1024] = { 0 };
  memcpy(re, data, size);

  char input_string[] = "Test␣string!";
  char *error_string;
  int error_offset;
  void *compiled_re;
  compiled_re = pcre_compile(re, 0,
                             &error_string,
                             &error_offset, 0);
  if (compiled_re == NULL) return 0;

  int result_vector[3];
  pcre_exec(compiled_re, NULL,
            input_string, strlen(input_string),
            0, 0, result_vector, 3);

  pcre_free(compiled_re);
  return 0;
}
```

Listing 1. Example of the `LLVMFuzzerTestOneInput` function, the user-defined entry-point for `libFuzzer` in our PCRE case study.

improve the performances and therefore the number of executions per second.

Once we have written the fuzzer entry point, we can recompile it with LLVM linking it to the translated module, ensuring that the `libFuzzer` instrumentation pass is executed on the LLVM IR before emitting the executable.

After performing all these steps, we run `rev.ng`'s fuzzer based on `libFuzzer` until it causes a crash in the program. Manually inspecting the binary at the identified location confirmed the presence of a stack overflow. Finally, by backtracking the bug to the original source code, we confirmed that it came from PCRE library and that it matched the CVE bug. This proved our points 1) and 2): `rev.ng` can be used with fuzzing to find vulnerabilities, even without source code.

The `afl` approach. To prove point 3), i.e., that using `rev.ng` with fuzzing lowers the time and the effort necessary for the analysis, we need to compare it to a similar workflow. To the best of our knowledge, `rev.ng` is currently the only tool able to provide coverage-guided fuzzing on user-

	Execs per second			Total execs
	1 min	10 min	60 min	60 min
`afl`	3 582	3 495	3 682	13 187 295
`rev.ng`	150 617	79 701	78 306	271 217 728

Table 1. Comparison of the performance of different fuzzing approaches for binaries without source code. `afl` is used in QEMU-*mode*, while `rev.ng` uses `libFuzzer`-based fuzzing on user-defined entry-point. The performances are reported in number of executions per second of the fuzzing target. Data are collected for 1, 10, and 60 minutes of execution.

selected entry-points without requiring the original source code of the program. For this reason there is not a direct candidate for performance comparison in fuzzing. Given that our use-case is for binary analysis in absence of source code, the obvious best candidate is `afl` in QEMU-*mode*.

In order to mimic the same entry-point for the fair comparison of fuzzing performance, we did not built the entire `less` program, otherwise `afl` would be too heavily penalized. Instead, we built a program similar to Listing 1 but directly linked against the PCRE library and reading data from standard input instead of using the `data` argument, and fed it to `afl` in QEMU-*mode*. We then executed `rev.ng`'s fuzzer and `afl` separately, to compare the fuzzing performance in terms of executions per second.

The results are reported in Table 1. Clearly the performances of `rev.ng`'s fuzzer are outstanding compared to `afl` in QEMU-*mode*, with speedups ranging from $20\times$ to $40\times$. The slight degradation in `rev.ng`'s speedups with time is due to the coverage-guided exploration progressively leading the fuzzer towards more complex inputs. This effect is not visible for `afl` for which the poor performance is due to other causes, discussed later. Despite the degradation, the performance of `rev.ng` still remains substantially superior to `afl`, allowing to explore a much larger set of inputs in the same time.

Many results in recent years have already shown the effectiveness of fuzzing [7], even specifically for the PCRE library (CVE-2017-16231, CVE-2017-7245, CVE-2017-7246, CVE-2017-7186). Note however that the exact time required to found a certain bug depends on the seed of the randomized algorithms used in the fuzzers, on the initial corpus of testcases provided as inputs, and other factors. For these reasons, in our experiments we did not measure the exact time necessary to find the vulnerability with `afl` and with `rev.ng`. What is important is the fact than `rev.ng`'s approach to fuzzing is much more efficient in exploring new input vectors, compared with previous state-of-the-art for coverage-guided fuzzing on user-defined entry-point on binaries without the availability of the source code.

In this respect, the poor performance of `afl` compared to `rev.ng` is to ascribe to two causes: first `afl` works by forking the fuzzed process multiple times, incurring in severe performance penalty for this; second, the fact that `afl` provides test vectors through files or standard input further affects its speed due to I/O latency.

The first problem could hypothetically be avoided using `afl`'s *persistent mode*, which minimizes the number of forks to reduce the performance penalty. However, *persistent mode* is not compatible with QEMU-*mode*, which makes it impossible to employ it for binaries whose source code is not available. This confirms that, for the binary analysis scenario, `rev.ng`'s fuzzer is by far the most efficient, bringing big improvements to the analysts workflows.

5. Conclusion and Future Work

In this work we showed that the `libFuzzer`-based fuzzing approach used by `rev.ng` is capable of identifying security vulnerabilities in real-world examples, even when the source code of the affected program is not available.

The binaries translated with `rev.ng` provide better performance than emulation with QEMU. In addition, the fuzzing approach used in `rev.ng` is much more efficient than the previous state-of-the-art for fuzzing binaries without access to the source code, represented by `afl`'s QEMU-*mode*. Finally, the approach adopted by `rev.ng` is naturally part of the reverse engineering workflow. When an analyst finds an interesting spot it can just mark the entry point and activate the automated fuzzing. Another key feature of `rev.ng` is that it is, at the best of our knowledge, the only tool capable of coverage-based fuzz-testing on user-defined entry-points that does not require the availability of the source code. All these features make `rev.ng` a perfect candidate for use in binary analysis tasks.

In the future, we plan to fully automate the fuzzing integration with the `rev.ng` framework, and to extend the integration also to LLVM-based symbolic execution engines like KLEE.

References

[1] A. Di Federico and G. Agosta, "A Jump-target Identification Method for Multi-architecture Static Binary Translation," in *CASES 2016*.

[2] A. Di Federico, M. Payer, and G. Agosta, "Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries," in *CC 2017*.

[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC, 2005.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO 2004*.

[5] Hex-Rays, "IDA Pro: About," https://www.hex-rays.com/products/ida/index.shtml.

[6] K. Serebryany, "Continuous Fuzzing with libFuzzer and AddressSanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, Nov 2016.

[7] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, 2018.

[8] M. Załewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[9] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[10] P. Hazel, "PCRE – Perl Compatible Regular Expressions," https://www.pcre.org/.

[11] CVE Details, "CVE-2015-3217," https://www.cvedetails.com/cve/CVE-2015-3217/.