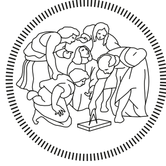


Dipartimento di Elettronica, Informazione e Bioingegneria
Master Degree In Computer Science and Engineering

Coverage-guided binary fuzzing with rev.ng and llvm libfuzzer

Master Dissertation of:
Antonio Frighetto

Advisor:
Prof. Giovanni Agosta

Supervisor:
Ph.D. Alessandro Di Federico

Co-Supervisor:
M.Sc. Andrea Gussoni

Academic Year 2019/2020

Contents

List of Figures **iii**

List of Algorithms **v**

List of Listings **vii**

List of Tables **ix**

Acknowledgments **xi**

Abstract **xiii**

Introduction **xvii**

1 Background **1**
- 1.1 LLVM . . . 1
  - 1.1.1 LLVM IR . . . 2
- 1.2 `rev.ng`: An Overview . . . 4
  - 1.2.1 Organization of the Lifted Code . . . 6
  - 1.2.2 Representation of the CPU State . . . 8
  - 1.2.3 Identification of the Function Boundaries . . . 9
- 1.3 Coverage-guided Fuzzing . . . 10
  - 1.3.1 Binary Fuzzing . . . 12
- 1.4 Exception Handling Mechanism . . . 13
  - 1.4.1 Exceptions in C++ . . . 14
  - 1.4.2 The Unwind Library Interface . . . 15
  - 1.4.3 Exception Handling in LLVM . . . 16

2 State Of The Art **17**
- 2.1 AFL in QEMU-*mode* . . . 17
- 2.2 Honggfuzz & QBDI . . . 18
- 2.3 Jackalope . . . 19
- 2.4 QASan . . . 20
- 2.5 Retrowrite . . . 21
- 2.6 Other works . . . 21

3 rev.ng-fuzz: The Fuzzing Framework **23**
3.1 Design 24
3.2 Implementation 26
3.2.1 Functional Correctness and Performance 26
3.2.2 Instrumentation 28
3.2.3 Generation of the Function Prototypes 31
3.2.4 Hooking 34
3.2.5 Sanitizers 36
3.3 Execution Flow 39
3.3.1 Compilation Stage 41

4 Make EnforceABI Great Again **43**
4.1 Isolation of Functions 43
4.1.1 Exception Handling in rev.ng 47
4.2 ABI Enforcement 49
4.2.1 Execute EnforceABI'd Code 51

5 Experimental Evaluation **57**
5.1 Reproducing CVE-2015-3217 57
5.2 Running a program with EnforceABI 61

Conclusions **63**

Bibliography **64**

List of Figures

1.1 Architectures supported by `rev.ng` 4
1.2 Overview of the static binary translation process 5
1.3 Representation of coverage-guided fuzzing 10
1.4 Strategy employed by coverage-guided fuzzing 11
1.5 Exception code flow in C++ 15
3.1 Overview of the fuzzing process 25
3.2 Comparison between the `root` and isolated functions 27
3.3 Transition back to `root` 28
3.4 SanitizerCoverage instrumentation 36
3.5 Representation of the AddressSanitizer shadow memory 37
3.6 AddressSanitizer instrumentation 38
3.7 Representation of the memory region for the emulated stack 40
3.8 Optimization pipeline 41
4.1 Pipeline with Function Isolation 44
4.2 Transition between the isolated realm and `root` 46
4.3 Basic block in `root` invoking its isolated function 47
4.4 Exception basic block 54
4.5 Example of execution flow with EnforceABI 56
4.6 CSVs commit state during stack unwinding 56
4.7 Pipeline with EnforceABI 56

List of Algorithms

3.1 The FuzzingReturn pass 29
3.2 The WrapperGenerator pass 32
3.3 The FunctionHooking pass 35

List of Listings

1.1 Sample C function and its corresponding LLVM IR 3
1.2 The root function 7
1.3 Example of raising an exception in C++ 14
1.4 Example of exceptions handling in LLVM IR 16
2.1 QEMU patch to enable AFL 18
2.2 QASan instrumentation of a TCG load instruction 20
3.1 The main function after fuzzing instrumentation 30
3.2 Example of a generated wrapper around my_func 33
3.3 Redefinition of rtb_malloc for hooking 35
3.4 Redefinition of rtb_malloc for ASan 38
3.5 The InitializeFuzzing routine 39
3.6 The LLVMFuzzerTestOneInput function 40
4.1 rev.ng helper responsible for throwing an exception 48
4.2 Comparison of a function before and after running EnforceABI . . . 50
4.3 Comparison of a call site before and after running EnforceABI . . . 52
5.2 Example of fuzzing functions before introducing rev.ng-fuzz 58
5.1 The LLVMFuzzingTestOneInput function for fuzzing less 59

List of Tables

5.1 Performance comparison between `rev.ng` and AFL 60
5.2 Performance comparison before and after running EnforceABI 61

Acknowledgments

I would like to thank Professor Agosta for introducing rev.ng to me, my supervisors, Alessandro for trusting me on this inspiring work and for his time and patience in providing me with precious suggestions for a good teamwork; Andrea for his calm and steady support, in particular during our interminable debugging sessions, and all the other good folks at rev.ng. I will always be very thankful.

I'm particularly grateful for my family, my parents and my sister, for their love, for being an inspiration, and for encouraging and supporting me along this challenging, yet rewarding, journey. A big thank to my longest and dearest friends for their moral support, for always being present and for the happy moments spent together. A special thank to my roommates for sharing good emotions during these hard times and bearing with me while working on this thesis. A sincere thank to all my friends with whom I spent these university years. I could have never accomplished it without you all.

Abstract

Software vulnerabilities remain one of the most significant threats challenging information security. Memory-unsafe programming languages such as C/C++ allow bugs that could be maliciously exploited in order to tamper with the system, exposing users to risk. Fuzzing is a widespread and effective software testing technique for discovering unknown security vulnerabilities, which consists in executing a program continuously with a large number of random inputs to cause a crash. Most state-of-the-art fuzzers are *coverage-based*, i.e., they leverage the code coverage to favour inputs that contribute at exploring new paths. The coverage is collected by instrumenting the program source code at compile time; and, intuitively, a higher code coverage can lead fuzzers to find more bugs. However, when it comes to fuzzing binary-only software (*source-unavailable*), compiler instrumentation is not possible, subjecting the field to further research.

`rev.ng` is a reverse engineering framework based on LLVM. It features a *static binary translation* tool that translates machine code into LLVM IR, a higher-level intermediate representation, independent from the input architecture, and suitable to perform a variety of analyses and transformations. From a security standpoint, it allows to instrument programs in order to analyze their behavior.

In this work, we propose a novel framework based on `rev.ng` and libFuzzer, the LLVM fuzz testing library, to perform coverage-guided binary fuzzing of executable programs. We show that our approach is fast, semantic-preserving and simply requires to implement the so-called *fuzz target*, a dedicated function that the fuzzer calls to pass its inputs to the interesting functions to fuzz, just as occurs for programs with source code available. We evaluate our framework on a real-world vulnerability and we show we manage to reproduce the bug successfully.

Sommario

Le vulnerabilità nel software continuano ad essere una delle minacce che la sicurezza informatica si trova ad affrontare. La mancata gestione automatica della memoria in linguaggi come C/C++ può causare errori di programmazione tali da essere sfruttati da malintenzionati per rubare dati sensibili o scalare i privilegi. In questo contesto, il *fuzzing* si configura come una tecnica di comprovata efficacia di ricerca di vulnerabilità, la quale consiste nell'eseguire un programma ripetutamente con input anomali al fine di causare un crash di quest'ultimo. I fuzzer più moderni fanno uso di *coverage*, ossia favoriscono input che contribuiscono a esplorare nuove aree del programma misurando quali zone di codice sono state eseguite finora. La coverage è raccolta con apposita instrumentazione del codice sorgente a tempo di compilazione, e intuitivamente, più alta è la coverage, più sono alte le possibilità di far sì che il fuzzer trovi bug. Tuttavia, quando si tratta di effettuare fuzzing su programmi di cui non si dispone del codice sorgente, instrumentare il codice non è più un'opzione fattibile. Ciò rende il fuzzing di software chiuso ancora area di ricerca.

`rev.ng`, un framework per attività di reverse engineering basato su LLVM, dispone di uno strumento che è in grado di tradurre codice macchina in LLVM IR: una rappresentazione intermedia ad alto livello indipendente dall'architettura d'origine, particolarmente adatta per svolgere analisi di programmi. Da un punto di vista della sicurezza, permette di instrumentare programmi al fine di analizzarne il loro comportamento.

In questa tesi, proponiamo un nuovo framework basato su `rev.ng` e su libFuzzer, la libreria di fuzzing parte di LLVM, per poter effettuare coverage-guided fuzzing di programmi di cui non si dispone del codice sorgente. Mostriamo come il nostro approccio sia veloce, preservi la funzionalità del programma originale e richieda semplicemente di implementare la funzione apposita per poter fare fuzzing, proprio come avviene per i programmi con codice sorgente disponibile. Testiamo il nostro framework su una vulnerabilità vera e propria e mostriamo come siamo in grado di riprodurre il bug con successo.

Introduction

Memory corruption vulnerabilities continue to pose threats against software written in memory-unsafe and type-unsafe programming languages such as C/C++, commonly used by operating systems as well as web browsers. Programs written in such languages may suffer from errors relating to the misuse of memory, including buffer overflows, use-after-free and type confusion issues. These memory violations are held accountable for many security breaches since an attacker may be able to exploit them in order to leak sensitive data or, worse, gain kernel code execution. While mitigations (e.g., ASLR, CFI [4]) raise the bar against attacks, exploitable bugs will continue to exist as long as these languages are used. As of now, such languages are not anywhere close to disappearing since extensively used in, e.g., performance-critical applications, embedded systems and resource-constrained environments.

To counter these issues, several effective approaches have merged in recent times. In particular, fuzzing and symbolic execution are two major software testing techniques for automatically generating test inputs and discovering vulnerabilities. Symbolic execution allows programs to take on symbolic input values rather than concrete values, and generates the test cases by solving the constraints for program paths through, e.g., a theorem prover. While effective, symbolic execution is known to suffer from the path explosion problem [9], since it triggers a large number of paths while solving the path conditions, making it impractical to employ it for large programs. By contrast, fuzzing executes a program recurrently by feeding it with random inputs so as to trigger a crash.

Coverage-guided Fuzzing

Modern fuzzers are *coverage-based*: they create new input seeds by mutating only interesting inputs, i.e., those that, in previous runs, contributed at exploring new parts of the program. If an input does not lead to traversing new paths, it becomes less favored in the next run. In order to monitor the progresses achieved during testing, these fuzzers keep track of which new paths are visited; and this information allows them to guide input generation for future executions. Thanks to this *feedback-driven* strategy, coverage-guided fuzzing has proved to be very successful in uncovering bugs with a severe security impact in largely used software.

In particular, American Fuzzy Lop (AFL) is a pioneer in this field [42]. It collects code coverage information to guide choosing inputs via compile-time instrumentation. In general, code coverage *per se* is a broad term to refer to measuring

which parts of a program have been executed so far. In practice, it can be quantified at the granularity of a basic block, edge, or path. While tracking accurate path coverage provides the most complete information, it is not feasible due to prohibitive instrumentation overhead. Thus, fuzzers balance coverage accuracy with performance. Indeed, AFL provides edge coverage tracking along with hit-count (i.e., how many times a branch is taken) for prioritizing some inputs over others.

Another cutting-edge coverage-guided fuzzer is LLVM libFuzzer [28]. It allows feeding fuzzed inputs to the library to test via a specific entrypoint, known as *fuzz target*. This function, whose implementation needs to be provided by the analyst, is linked against the library and is repeatedly called with each new input generated from the mutated seed corpus. To enable libFuzzer and its instrumentation, it suffices to compile the source code with a specific flag. As a result, in principle, libFuzzer requires the source code to work.

While fuzzing source-available programs is a well-established practice area, assessing the robustness of binary-only software through coverage-guided fuzzing is still an open challenge. Existing works primarily rely on *dynamic binary instrumentation* [13] (e.g., AFL in QEMU-*mode*) in order to collect the code coverage. This is straightforward, but may incur very significant overhead. As we are going to see, our approach is significantly different from any previous work and aims at offering fast performance in an architecture-independent way.

Static Program Analysis

Program analysis is an effective approach to recover useful information from closed-source, legacy and commercial off-the-shelf (COTS) binaries. Such information can help the analyst understand the behavior of a program, to evaluate it, test it, and even harden it by retrofitting security features (e.g., CFI) directly into the program. In particular, the *static program analysis* aims at discovering semantic properties of programs without running them, as opposed to *dynamic program analysis*, which operates during program execution.

Understanding if a program will run as intended means checking if this program satisfies a behavioral property of interest. Examples of properties that static analysis aims at reasoning about include *safety* properties, e.g., proving absence of runtime errors, and *liveness* properties, e.g., proving program termination. Since any interesting semantic property is undecidable (per Rice's theorem), static analysis computes an approximation of the program behavior, meaning that it may not always be able to provide an exact answer. However, if an analysis answers with *yes* or *no* in the majority of program instances, and, only infrequently, *don't know*, it is still of great usefulness, as long as the computed information is correct and the analysis always terminates. Indeed, it turns out that, besides being of large interest, static analysis is widely employed, as it strengthens compilers, program verification and software inspection tools, and code optimizations.

Our Design

`rev.ng` is a unified static program analysis framework based on QEMU, an emulator [7], and LLVM, a robust compiler infrastructure [29]. Some of the challenges

that `rev.ng` attempts to solve include accurate program control-flow graph recovery and function boundaries detection in an architecture-independent way. Its core component is a tool that performs *static binary translation*, i.e., given an executable program that targets a certain architecture, it lifts the program into QEMU tiny code, the intermediate representation employed by QEMU, which abstracts away all architecture-specific details. Then, it is translated into another intermediate representation, the LLVM IR, which is particularly amenable for analyses and transformations. From a security perspective, the LLVM IR can be used to instrument programs in order to get deeper insights on their behavior.

In this work, we propose the following contributions.

`rev.ng-fuzz`. We propose a novel framework based on `rev.ng` and LLVM libFuzzer in order to perform coverage-guided binary fuzzing of executable programs, starting from user-defined entry points. This is particularly beneficial, when, for example, after a reverse engineering effort, the analyst has identified a function with user-controlled arguments. We show that our approach leverages *static binary instrumentation* to collect code coverage, incurs close-to-native overhead, is semantic-preserving and simply requires to implement the fuzz target, the function that guides the fuzzing process, as occurs for programs with source code available.

`EnforceABI` Improvements. We also propose a novel approach to allow the ABI enforcement on the LLVM IR module produced by `rev.ng`, by promoting the variables employed to represent the state of the processor, from LLVM global variables to local variables and function arguments. This has been primarily achieved in order to improve the fuzzing rates, since such a transformation allows to catch more compiler optimization opportunities. Our effort encompasses the design and implementation of a fail-safe mechanism for safeguarding the execution in case of errors related to the control-flow recovery inaccuracies due to static analysis intrinsic limitations.

Thesis organization

A brief overview of how the thesis is structured follows.

- Chapter 1 lays the necessary background for understanding this work.
- Chapter 2 provides detailed insights into state-of-the-art coverage-guided binary fuzzers in order to make comparisons with `rev.ng-fuzz`.
- Chapter 3 introduces `rev.ng-fuzz`, our coverage-guided binary fuzzer: we discuss its design, implementation, and the execution flow of the fuzzer.
- Chapter 4 deals with the enhancements over the ABI enforcement analysis in order to improve the fuzzing rates.
- Chapter 5 is devoted to evaluate `rev.ng-fuzz` on a real-world vulnerability and to compare its performance against another popular coverage-guided binary fuzzer. We also show preliminary results when running an enforced ABI binary.

Finally, we draw the conclusions of our contributions and review what has yet to be done.

Chapter 1

Background

This chapter lays the groundwork required to have a full understanding of this thesis work. We start off by introducing LLVM, a robust open-source compiler infrastructure and QEMU, a system emulator, since they comprise the mainstays of `rev.ng`. The core component of `rev.ng` is a static binary translator and it is analyzed hereafter. As it underpins our work, it is rather important to understand how it works. Then, we cover the concepts behind fuzzing, a software testing technique aimed at discovering bugs. We conclude by dissecting the exception handling mechanism, since it is extensively used in the second part of the work.

1.1 LLVM

LLVM (previously Low Level Virtual Machine) is a comprehensive compiler infrastructure for code generation and low-level optimizations [29]. The project was initially started by a team of researchers led by Chris Lattner at the University of Illinois at Urbana-Champaign, around 2000; afterwards, the team was hired by Apple Inc. LLVM was devised with strong compiler theory as well as a clean separation of components. Thanks to its simplicity in the design and implementation, it has been gaining popularity since its open-source release, and is being widely used both in research and industrial compilers. The compiler is structured in a three-tier architecture, made up of language-dependent frontends, an intermediate code optimizer, and a target-independent code generator, which can create binaries for several types of target CPUs. Such components, designed to be loosely coupled among each other, work as follows.

Frontend. The frontend is given a source code file as input and parses it in three stages: i) *lexical analysis*, whose role is to split the source code into substrings, called tokens and preprocess them; ii) *syntax analysis*, where the tokens are parsed to identify the structure of the program, typically with a bottom-up approach, in order to create a syntax tree and iii) *semantic analysis*, which fills the syntax tree with semantic information (e.g., static type checking) and builds a symbol table. The resulting abstract syntax tree, being still language-specific, is translated into the LLVM target-independent intermediate representation language, known as

LLVM IR. The most popular and modern LLVM frontend is Clang, a frontend for C, C++ and Objective-C. Other examples of programming languages that have frontends built on LLVM include Rust and Swift.

Optimizer. The unoptimized LLVM IR generated by the frontend is passed to the optimizer, which can transform and analyze the IR multiple times, while preserving its semantics. It selectively applies several optimization techniques to achieve different goals, e.g., to increase the performance or reduce the code size. The simplest ones consist in e.g., removing dead-code, propagating constants and coalescing multiple instructions into simpler ones; going towards more advanced loop-based transformations and interprocedural optimizations.

Backend. The optimized IR is passed to a target-independent code generator which produces executable code for several types of CPUs, such as x86-64, ARM, PowerPC, MIPS and SPARC. Some of the tasks that the code generator needs to deal with concern machine-dependent transformations, including instruction selection, instruction scheduling and register allocation.

Such a separation of concerns not only allows each component to focus on a specific task, but also avoids rewriting the entire pipeline if a new architecture is to be supported, *de facto* requiring to maintain only a new backend, thus reducing the implementation complexity. Furthermore, it is relatively easy to add new analyses and transformations, better known as *passes*, making the framework modular.

1.1.1 LLVM IR

At the core of the LLVM framework, there is its well-known intermediate representation, a low-level strongly-typed RISC-like virtual instruction set, which is the form internally used to represent the program being compiled, emitted by the frontend. Like a RISC architecture, memory is accessed solely through load/store instructions and all the other instructions have registers as operands, but there is no bound in the number of virtual registers that can be used. There are several advantages in passing through a representation that is independent of the original language the source code was written into, instead of a one-to-one compilation to binary code; including, architecture-independent optimizations as well as the possibility of targeting multiple processors at once. Thus, while being close to assembly languages, LLVM IR abstracts away any architecture-specific detail.

In LLVM IR, a translation unit of the input program, i.e., its content, is called *module*. A module is composed of a sequence of *functions* and *global variables*. A function can house *basic blocks*, which, in turn, contain one or more *instructions*. Each basic block must terminate with a *terminator instruction* (e.g., a branching instruction). One of the characteristic of LLVM IR (common to most intermediate representations) is the SSA (Static Single Assignment) form, where each register is assigned exactly once. Hence, a special instruction was introduced to allow the same variable to take different values, namely, the `phi`-instruction, which sets a variable to a value based on the control flow path taken. Besides enabling several optimizations, such form significantly simplifies the navigation of the values that an instruction uses and makes *use-def* chains explicit: in fact, each user has a single definition.

```c
int function(int *);

int main(int argc, char *argv[]) {
  int res;
  int local_var = 2;

  if (argc > 4)
    res = 1 + function(&local_var) + argc;
  else
    res = 1;

  return res;
}
```

```llvm
declare i32 @function(i32*)

define i32 @main(i32, i8**) {
entry:
  %2 = alloca i32
  store i32 2, i32* %2
  %3 = icmp sgt i32 %0, 4
  br i1 %3, label %true, label %epilogue

true:
  %4 = add i32 %0, 1
  %5 = call i32 @function(i32* %2)
  %6 = add i32 %4, %5
  br label %epilogue

epilogue:
  %7 = phi i32 [ %6, %true ], [ 1, %entry ]
  ret i32 %7
}
```

Listing 1.1: Example of a C snippet and its corresponding LLVM IR. `%2` corresponds to `local_var`, which, unlike `res`, was not promoted to SSA value since its address is passed to another function. `local_var` has been initialized to `2` through the `store` instruction. The `if` statement consists of a `icmp` instruction (where `%0` represents `argc`) and a conditional branch. If the `true` branch is taken, the return value is computed through two additions and the result of a function call. The call to `function` has as unique argument a pointer to `local_var` (`%2`). Eventually, the value to return, `%7`, is chosen by means of a `phi` instruction: depending on which basic block we are arriving from, the value `%6` or the constant `1` is employed.

1.2 rev.ng: An Overview

rev.ng is a reverse engineering framework based on QEMU and LLVM, focused on static program analysis [14][17]. Originally developed by Alessandro Di Federico as part of his Ph.D. research, it is now maintained by the homonymous company. rev.ng has been designed with the ultimate goal of facilitating reverse engineering tasks by providing a unified framework for static and dynamic analysis, which employs a set of algorithms and principled techniques that work independently of the architecture of the analyzed program. Hence, rev.ng's analyses are completely agnostic with respect to the input architecture. The core component is a tool that specializes in performing *static binary translation*, a technique that allows software compiled for an architecture (e.g., ARM) to be converted to run on another architecture (e.g., x86-64), achieving reasonable performance.

rev.ng takes as input a program for Linux, macOS or Windows, loads it into memory and parses it. It starts by separating the code and data segments and identifying the starting addresses of the basic blocks (known as *jump targets*) of the original program. Once the entrypoint and the exported functions are found, the discovery of new jump targets proceeds iteratively: the program's global data is scanned looking for pointer-sized values, i.e., addresses pointing to executable area. This typically include function pointers, C++ virtual tables and branch tables, which reside in read-only memory. During the translation, destination addresses of direct jumps and targets of branches materialized into registers (*literal pools*) are collected as well.

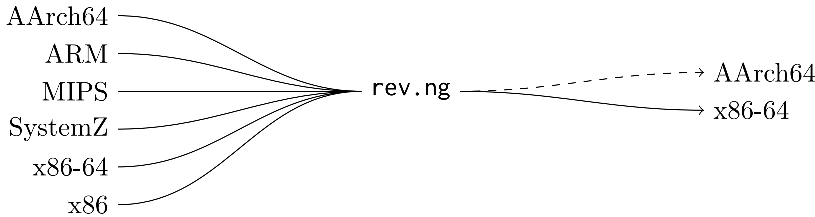


Figure 1.1: Architectures supported by rev.ng. On the left, there are the input architectures, on the right, the output ones. Support for AArch64 will be added soon. Note that, any architecture supported by QEMU is virtually supported by rev.ng too, i.e., the engineering effort of adding a new ISA is mostly straightforward.

QEMU. The actual interpretation of the *instruction set* (ISA) is decoupled from the control-flow graph recovery, and, in order to handle different architectures, rev.ng relies on another existing abstraction layer: QEMU, a machine emulator [7][30]. QEMU emulates a full operating system including processors, memory management units, peripherals, etc., thus, for example, it can run x86 guest OSes on a ARM host. To achieve this, QEMU features a *dynamic binary translator* that converts binary code from an architecture target to the host architecture. During program execution, it incrementally translates basic blocks into sequence of instructions that can be executed natively by the host processor. One interesting thing

about QEMU is that it tackles the challenge of translating code not by directly generating native code, but instead passing through an hardware-independent intermediate representation, alike to a compiler's one. Hence, the QEMU translation engine, known as *tiny code generator* (TCG), consists of two main parts: a *frontend*, which disassembles guest instructions and converts them into fewer simpler RISC-like instructions (called *micro-operations*), caching the translated blocks for future executions; and a *backend*, which turns such micro-operations into executable code.

Now, after harvesting all the possible *jump targets*, `rev.ng` employs the QEMU's frontend to translate the basic blocks at the corresponding address of the *jump targets* previously found, into TCG micro-operations. These sequence of TCG instructions are then expanded into LLVM IR instructions, which are all collected to form a LLVM recompilable module. The result is a new binary, with all the segments of the original program located at their original position, plus an extra segment containing the translated code (i.e., the lifted code that will be executed) which will reside in a different location in memory. Figure 1.2 provides a representation of the overall translation process.

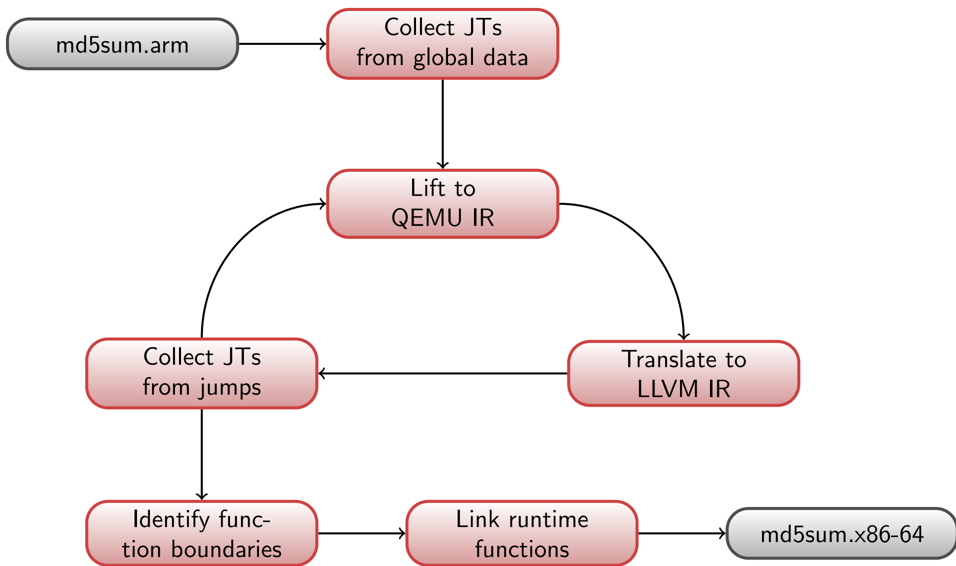


Figure 1.2: An overview of the *static binary translation* process performed by `rev.ng`. The input binary is loaded into memory, and a pre-processing phase parses the segments and harvests an initial set of *jump targets* from the global data and the program's entry points. Then, new jump targets are discovered iteratively: the code at the address corresponding to each jump target is first translated into TCG instructions and, subsequently, into LLVM IR, which is analyzed to recover additional jump targets. If new jump targets cannot be found, further analyses are carried out (such as, the one that aims at detecting the function boundaries). After lowering it down to object code, it is linked against the necessary libraries, recompiled to the target architecture, and finally, the new executable is generated.

While undertaking the complex task of supporting multiple instruction sets in a unified manner by resorting to QEMU (precisely, its ISA-independent intermediate representation) has enormous benefits – such as avoiding the large amount of work required to handle multiple architectures and integrate new ones –, different challenges arise. In fact, any subsequent analysis that is built on this representation cannot make assumptions about the original architecture. This implies initial lack of knowledge of what a function call or a return instruction are, whose concepts need to be redefined in an architecture-agnostic way. For instance, when dealing with the call instruction, we may ask ourselves: is the return address spilled onto the stack (like in x86) or stored in a dedicated register (as it happens in most RISC architectures)? Therefore, `rev.ng` is designed to focus on the features exposed by the instruction set, and not on the instruction set itself. Figure 1.1 shows the current supported architectures by `rev.ng`.

Note that, since `rev.ng` makes use of the *user-mode* emulation of QEMU, meaning that only the CPU is emulated and not the entire hardware, system calls are forwarded to the host kernel, whereas vector and floating point instructions are handled via software helpers (e.g., the floating point division is simulated with integer instructions). Also note that, despite using QEMU which *per se* operates at runtime, `rev.ng` remains a static binary tool since the translation happens offline (the program is never executed during the lifting process).

In the following sections we take a deeper look at how the lifted `rev.ng` module is organized, how `rev.ng` represents the state of the CPU, and how the process of identifying the original functions occurs.

1.2.1 Organization of the Lifted Code

During the lifting process, the input basic blocks of the program are converted into QEMU translated blocks and later into LLVM basic blocks. Note that an original instruction can be translated into multiple TCG micro-operations. This actually occurs for almost any instruction, since it typically comes down to i) a load from guest register(s) to QEMU temporary variables, ii) the actual operation, and iii) a store from the temporary back to the guest register(s). In this regard, TCG micro-operations are said to have no side effects, meaning that they exactly perform what they are meant to do. For instance, a guest conditional move instruction will be expanded into multiple TCG instructions (minus the TCG micro-ops post-optimizations). Consequently, the original instructions will correspond to multiple LLVM IR instructions.

Lifted code. All the translated code is organized in form of LLVM basic blocks and it is congregated in a single LLVM function, known as `root`. Note that, at this stage, there is no concept of functions of the original program, given that they have been grouped in basic blocks in `root`. Thus, when a function call is identified, an ad-hoc function marker (`function_call`) is emitted, which takes the callee basic block as one of the arguments. As a result, besides containing all the code that will run at program execution, the `root` function is also responsible for transferring the control flow from one basic block to another when, for example, in presence of an indirect function call. Such a flow transfer is implemented through a *dispatcher*.

```
define void @root(i64) {
dispatcher:
  %0 = load i64, i64* @pc
  switch i64 %0, label %abort [
    i64 4198400, label %bb._start
    i64 4198410, label %bb.main
    i64 4198418, label %bb.main.0x11
    i64 4198422, label %bb.bar
  ]

bb.main:
  ; 0x40100E: call 0x401016
  %1 = load i64, i64* @rsp
  %2 = sub i64 %1, 8
  %3 = inttoptr i64 %2 to i64*
  store i64 4198418, i64* %3
  store i64 %2, i64* @rsp
  store i64 4198422, i64* @pc
  call void @function_call(i8* blockaddress(@root, %bb.bar),
                           i8* blockaddress(@root, %bb.main.0x11),
                           i64 4198418)
  br label %bb.bar

bb.main.0x11:
  ; ...

bb.bar:
  ; ...
}
```

Listing 1.2: The `root` function containing the *dispatcher* and all the identified basic blocks in the original binary. The dispatcher consists of a `switch` statement, which maps each *jump target* address to the corresponding LLVM basic block. It is mandatory to transfer properly the execution when the basic block target could not be identified. Thus, depending on the value of the program counter (represented by the `pc` global variable) at runtime, the appropriate basic block is taken. If no match is found, a dedicated basic block (`abort`) handles such a situation. The snippet also shows the basic block belonging to `main`. Particularly, the IR exemplifies the original disassembled x86 instruction calling a function at address `0x401016`. The first instructions push the return address onto the top of the stack. Then, the program counter is updated and a branch to the function `bar` is performed (as a last instruction). The `function_call` marker serves to denote that, originally, a function call was taking place and has no functional purposes. The marker takes as input the entry basic block of the function, the return (*fall-through*) basic block, and the return address. Finally note that, if the binary has not been stripped of symbols, basic blocks appear with their original name.

Dispatcher. The *dispatcher*, a key component in `rev.ng`, consists in a jump table with all the possible basic blocks enumerated. Its task is to route the control flow to the appropriate basic block at runtime in case the target location to jump to could not be resolved statically (e.g., in indirect calls). In practice, as Listing 1.2 shows, the dispatcher is implemented as a `switch` instruction inside `root`, which tests the value of the program counter at runtime against all the addresses of the identified basic blocks, and divert the execution to the correct basic block.

While it would be desirable to use the dispatcher only for indirect function calls, it may be employed for indirect branches as well. In order not to damage the performance of the recompiled code, the less the dispatcher is taken, the better. In fact, if too dense, the worst case implementation has a cost that is logarithmic in time (binary search). That said, the compiler will attempt to lower the `switch` statement down to a lookup table, which has a complexity that is constant in time.

Finally note that, on some architectures (e.g., x86), when possible, QEMU tries to avoid inspecting the emulated program counter by patching the current translated block so that it directly jumps to the next translated block to be executed (*direct block chaining*). `rev.ng` applies the same concept, which consists in devirtualizing an indirect function call, i.e., emitting an inline `switch` statement with only a subset of all the basic blocks targets enumerated, namely, those targets that could be computed statically. This way, jumping back to the dispatcher can be avoided.

1.2.2 Representation of the CPU State

Binary translation needs to face the problem of modeling the CPU via software, in order to emulate it. QEMU achieves it by describing the CPU as a C variable. For example, the ARM architecture is modeled through a dedicated structure (`struct CPUARMState`), which includes an array representing the general-purpose registers, a bitmask for the saved program status registers (SPSR), separated flags for the current program status register (CPSR), and so on. Conversely, `rev.ng` tackles this issue by modeling each individual state of the CPU (register or flag) through LLVM global variables, known as *CPU State Variables*, or simply CSV. For example, when translating to x86-64, the LLVM module shall provide global variables representing the usual ABI registers, including general-purpose registers, floating-point registers, stack pointer (`rsp`), program counter (`pc`), and so forth. As a result, this means that every instruction reading or writing a register (via `load`/`store` instructions in LLVM) is performed on a global variable.

Although such an approach may be deemed as not beneficial at first glance – as global variables are difficult for the compiler to optimize, since the compiler must assume that every function in the program could modify it and every pointer of appropriate type may point to that global variable –, keep in mind that: first, this problem, as aforementioned, is intrinsic to any binary translation tool; secondly, the compiler will still be able to remove several load and store pairs (*whole-program optimization*). In Chapter 4, we will show how, after recovering more abstractions such as functions identification and their interface, it is possible to promote registers (in form of global variables) to formal parameters and local variables of LLVM functions; and how we have been able to execute such a translated binary.

1.2.3 Identification of the Function Boundaries

rev.ng includes a dedicated analysis that aims at detecting the function boundaries of the original program. Intuitively, once that the function entry points have been identified, they can be associated with all their reachable basic blocks, skipping over the call sites, so that a chain of basic blocks, i.e., a function, is built. While it may seem rather straightforward, achieving an analysis of this kind that works across multiple architectures is non-trivial. First, note the high correlation between the function boundaries detection and the control-flow graph recovery: the accuracy of the detected start and end addresses of a function strongly depends on the quality of the recovered control-flow. This implies that, if the CFG misses information about the destinations of an indirect call, or includes a limited and wrong subset of destination basic blocks, the real destination may not belong to the function, thus resulting in an accuracy loss. Secondly, note that, rebuilding a complete control-flow graph that matches the original binary one's is reducible to the Halting problem (i.e., not computable) [10][26] – nonetheless, it is possible to formulate rigorous analyses in the majority of the problem instances.

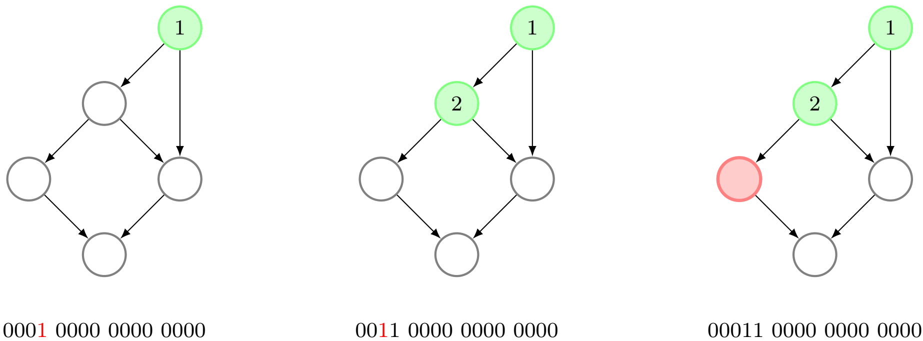
More specifically, the function boundary identification comprises the following steps. An initial set of entry points basic block is gathered, referred to as *candidate function entry points* (CFEPs). Function calls represent the most well grounded source among the possible CFEPs, since they are destination of a jump, but not all functions are explicitly called. Some functions may be called indirectly or be never used. Thus, unreferenced global data, which include function pointers and C++ virtual tables, are collected as well. After computing a preliminary set of CFEPs, for each candidate entry point, the CFG is traversed in order to identify the reachable basic blocks. When a function call is reached, it is not followed; instead, the exploration continues until a jump to the *return address* is encountered. Note that we previously lost the concept of what a return instruction is, in order to be ISA-agnostic, therefore the analysis is further refined by establishing whether the last indirect branch instruction is actually a return, a tail call, etc., and what the kind of function is (a regular one, a function that does not return, a function leaving the stack frame in a wrong state upon returning, etc.). Eventually, a set of possible functions is provided.

1.3 Coverage-guided Fuzzing

Fuzzing is a widely adopted technique for discovering software security vulnerabilities, and, in its simpler form, consists in automatically generating some inputs, randomly mutating and feeding them to a target program, in the hope of triggering bugs [39]. Around 2014, fuzzing experienced unprecedented advancements, with the introduction of AFL (American Fuzzy Lop) by Michał Zalewski, and it has been gaining popularity steadily ever since [42]. Compared to its more primitive form, what makes AFL innovative is its *coverage-based* approach, which prioritizes inputs that lead to traversing previously unexplored execution paths, as illustrated in Figure 1.3. In order to measure the coverage, i.e., to trace path transitions exercized by the random inputs, the program is instrumented at compile-time. This coverage information is used to drive the fuzzing strategy, and the greater the coverage, the higher the chances of finding bugs.

```c
int main(int argc, const char **argv) {
  if (argc > 4) {  // (1)
    if (argv[0][1] == 'A')  // (2)
      bug();
    else
      do_stuff();
  }
  return 0;
}
```

(a) Sample C program with a bug



(b) Control-flow graph of some states of the program during execution of random inputs

Figure 1.3: A representation of *coverage-guided fuzzing*. The program is tested with some random input, represented by the sequences of bytes below. It turns out that, for specific inputs, the program leads to a crash due to a bug in the code. The control-flow graphs presented above show the exact states leading to the bug. Such states may have been executed after different runs of the program. Note the coverage-based approach: the input favouring condition `(1)` is preserved among the corpus of inputs, since it led to a previously unexplored path. After random mutations, the fuzzer explores new paths, including `(2)`, which triggers the bug.

Then, at run-time, new inputs are continuously generated by the fuzzer through a feedback-mechanism; however, only those inputs that maximize the coverage are retained. While most fuzzers generate test inputs through mutation of inputs, few others are *grammar-based* [36], i.e., they follow some pre-defined grammar in order to generate syntactically correct inputs for the target binary (e.g., when fuzzing parsers), in an attempt to go deeper in the code exploration.

AFL. In its design, AFL is heavily inspired to *genetic algorithms* for generating test cases. Given an initial pool of user-supplied inputs (seeds), AFL moves them into a queue (①). Then, a sample is selected, and it is manipulated through genetic operators, i.e., the inputs are first combined and exchanged, and secondly they are mutated with different techniques, e.g, bit flipping, addition and replacement of bytes, etc (②). Such new inputs are sent to a fork-server, which spawns child processes, and executes them with the mutated inputs. During the execution, AFL monitors the testing program and collects the coverage information via instrumentation (③). If the fuzzer manages to explore new paths, the sample, being more promising, is privileged for future mutation; whereas the test cases that do not increase the coverage are discarded. The objective is met when the target program crashes, leading the fuzzer to generate a report (④).

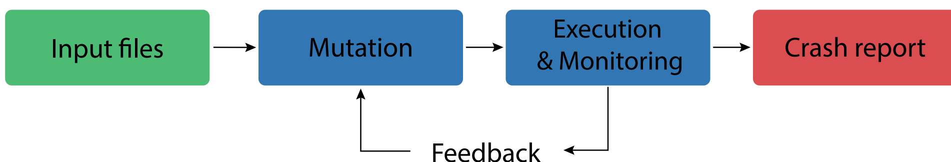


Figure 1.4: The *feedback-driven* strategy leveraged by coverage-guided fuzzers.

In order to filter out uninteresting inputs, AFL tracks approximate edge coverage, i.e., it records which branches are taken and the number of times by injecting the following instrumentation to the target program through custom Clang/GCC wrappers:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

AFL represents an edge as the XOR between the current basic block, which is assigned a random constant, and the previous basic block, i.e., as the tuple `(prev_location, cur_location)`. The coverage information is stored in a compact structure, known as tracing bitmap, a 64KB hash table shared between AFL and its target process via POSIX `shmget` routine. When an edge is visited, the hit is recorded by incrementing the value in the bitmap corresponding to the hash of that particular edge. Due to the size of the data structure employed, there may exist sporadic collisions.

LLVM libFuzzer. Another popular state-of-the-art coverage-guided fuzzing engine is LLVM libFuzzer. Originally written by Konstantin Serebryany [28], libFuzzer is directly linked against the library to test. It starts with a corpus of inputs,

picks a random sample, which is randomly mutated, and feeds it into the target program through the so called *fuzz target*, also known as harness (`LLVMFuzzerTestOneInput`), shown below, which is implemented by the user.

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
 DoSomethingInterestingWithMyAPI(Data, Size);
 return 0;
}
```

Unlike AFL, which is designed to fuzz standalone programs that typically accept input from standard input, libFuzzer is in-process, i.e., it executes the fuzzing function many times with different inputs within the same process. This allows libraries and their APIs to be fuzzed rather easily, as long as one understands the APIs being fuzzed. When it comes to fuzzing routines that accept more complicated inputs (e.g., system calls), where generic mutations may fail to produce valid bytes to pass input checks (thus resulting in being discarded), libFuzzer allows to implement custom mutators to improve the mutation quality, i.e., specialized methods that permit to specify a grammar for certain inputs (e.g., via protobuf messages [38]). With regard to this, libFuzzer is more tailored to *structure-aware fuzzing*. libFuzzer relies on a LLVM module pass, SanitizerCoverage (SanCov) [3], to statically instrument the program for coverage tracking.

Additionally, existing fuzzers often leverage compiler *sanitizers* [37] to enhance bug detection, i.e., error detector tools that work by injecting further checks into the program. When a crash occurs, sanitizers provide a log including the bug and the stack trace of the program. In particular, AddressSanitizer (ASan) can detect memory corruption errors, including buffer overflows and use-after-free issues, and it is based on compiler instrumentation and a shadow memory region that encodes the state of the program (more on AddressSanitizer in Section 3.2.5). Indeed, AddressSanitizer is particularly beneficial when paired with fuzzing, since it may trigger bugs that would silently be missed otherwise.

1.3.1 Binary Fuzzing

While whitebox (*source-available*) fuzzing is a relatively consolidated research area, assessing the security of blackbox (*closed-source*) software (*binary fuzzing*) is still largely an open challenge. Existing works primarily resort to dynamic binary translation (e.g., AFL in QEMU-*mode* [42], VUzzer with PIN [35], Honggfuzz & QBDI [25], more in Chapter 2), which injects the instrumentation to enable fuzzing during program execution. Despite being straightforward, this process incurs significant overheads and, consequently, a reduced effectiveness, since the coverage of the input space exploration may be restrained. Other works achieve binary fuzzing through static binary rewriting [40] or reassembleable assembly [41]. While effective, these approaches are oftentimes built on a set of heuristics that depend on characteristics of the binary and the platform, thus targeting only a specific class of them.

Unlike the previous works, the framework we aim at presenting in Chapter 3 leverages static binary translation, and employs LLVM libFuzzer as mutational engine to achieve fuzzing of binary-only software with near-native overhead.

1.4 Exception Handling Mechanism

In this section, we review the exception handling mechanism, which defines the program behaviour after a runtime error (*exception*) has been detected. Besides being of utmost importance to support exceptions when lifting a binary, `rev.ng` also leverages exception handling to correct the execution in case of unusual control-flow inaccuracies, as we are going to see in Chapter 4. Hence, it may be helpful to become familiar with such a mechanism beforehand.

The exception handling mechanism is designed to allow to recover from unexpected, possibly rare, conditions (e.g., memory shortage, missing file, etc.) that can occur during an application's lifetime. Raising an exception affects the control-flow: the execution is transferred to a registered routine, typically a handler, which, if provided, can manage the anomalous event and, eventually, return gracefully.

There are two implementations for handling exceptions.

Setjmp/Longjmp Exception Handling. This mechanism provides a non-local jump facility to suspended stack frames. The `setjmp` routine saves the current execution context into a platform-specific structure (`jmp_buf`), whereas `longjmp` performs the non-local transfer to the place originally set by `setjmp`. While easy to implement, this approach has some intrinsic issues. First, allowing to branch to almost anywhere in the program may quickly lead to program flow disruption. Secondly, it is rather expensive, since the cost of saving a program state is always paid, regardless of whether an exception was raised or not.

Itanium ABI Zero-cost Exception Handling. This mechanism allows control-flow transfers from the point where an exception is raised to a handler. When the handler completes, the execution continues as though no interference occurred with the application's main flow. Unlike the previous approach where the compiler is unaware of exception handling, such a methodology uses tables generated by the compiler together with the machine code. Since the overhead of throwing an exception is incurred only when an exception is actually raised, this mechanism is known as *zero-cost exception handling*.

The last approach is based on the specifications defined by the Itanium ABI [1], and *de-facto*, it is the standard C++ ABI employed on many architectures [12]. In fact, unlike other core features of C++, exception handling is strongly tied to the platform and described by the platform's ABI.

As an example of its usage, consider the snippet shown in Listing 1.3. In C++, the code that may generate an error is enclosed in a `try-catch` block. When attempting to divide by zero, an exception is raised via the `throw` statement, caught and finally logged.

1.4.1 Exceptions in C++

Throwing an exception requires *unwinding* the call stack until an exception handler is found, i.e., the stack frames between the throwing and catching function are removed. Besides being assisted by the compiler, the C++ runtime library (`libc++abi`) provides support for handling exceptions with the aid of a platform-specific *unwind library*, which performs the actual stack introspection and unwinding. Historically, Linux distributions have been using `libgcc` as unwind library, whereas macOS and FreeBSD have come with LLVM `libunwind` for a while.

When an exception is raised, the runtime library allocates an *exception structure object* and passes it along to the unwind library, which searches for the appropriate handler so that the exception can be managed. In fact, for a given `try`, there may exist several `catch` statements that must be examined to see if they should catch the exception. Determining which handler to call, i.e., the one that matches the thrown type, is accomplished by the *personality routine*, a language-specific callback which consults the unwind tables generated at compile-time to find information about the pertinent handler. Such tables are found in sections `.eh_frame` in ELF binaries and contain information about the activation records that must be unwound while processing an exception. Specifically, within these tables, the personality routine examines the *language-specific data area* (LSDA), which, among other things, contains a list of entries indicating, for a given call-site that may throw, its corresponding handler. The control is then transferred to it. If no entry is found in the LSDA, `std::terminate` is called, causing program termination.

```
int divide(int num, int den) {
  if (den == 0)
    throw std::runtime_error("Attempting to divide by zero");
  return num / den;
}

int main(int argc, char *argv[]) {
  int res = 0;
  try {
    res = divide(argc, 0);
    std::cout << "The quotient is: " << res;
  } catch (std::runtime_error &e) {
    std::cout << "Exception occurred: " << e.what();
  }
  return res;
}
```

Listing 1.3: Example of raising an exception in C++. The `try`-block surrounds statements in which an exception is thrown by the function `divide`. The `catch`-block, instead, contains code that is executed after throwing the exception. In particular here, we log the string that identifies the exception. Note that the `throw` statement can be placed anywhere in the code, not necessarily within a `try`-block.

1.4.2 The Unwind Library Interface

In this section we explore the life of an exception, starting from its throwing, passing through the unwind library interface, till reaching its catching. The journey can be summarized as follows.

1. `throw` statement. The compiler translates the C++ `throw` statement into a call to `__cxa_allocate_exception`, which allocates an exception object, along with a call to `__cxa_throw`, which starts the events involved in processing the exception by calling into the unwind library. The latter call never returns: either the exception is caught appropriately or `std::terminate` is called.

2. `_Unwind_RaiseException` routine. `__cxa_throw` passes the exception object to `_Unwind_RaiseException`, which takes control and performs the stack unwinding with the help of the personality routine (`__gxx_personality_v0`). It is achieved in two different stages: by calling `unwind_phase1` first (search phase), and `unwind_phase2` (cleanup phase) afterwards.

- **The search phase**. Each active stack frame is inspected by calling the personality routine with `_UA_SEARCH_PHASE` flag in order to find the matching handler. Recall that the stack can be walked back thanks to the call frame information returned by the LSDA. If the handler is found, the personality routine returns `_URC_HANDLER_FOUND` code, otherwise `_URC_CONTINUE_UNWIND` code is returned, which warns to continue searching for the handler.

- **The cleanup phase**. Once the correct exception handler is identified, the unwinder walks back the stack frames again, this time however, the personality routine is called with `_UA_HANDLER_FRAME` flag in order to remove each stack frame, thus actually unwinding the stack. This includes cleaning up local objects that are going out of scope by calling their destructor. When the object's destructor completes, the control is returned to the unwinder through `_Unwind_Resume`.

3. Landing pad execution. Eventually, after the stack frames removal, the control is transferred to the matching `catch`-block (*landing pad*) and is executed.

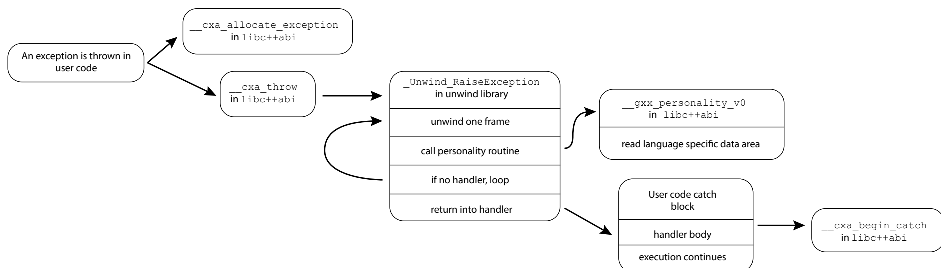


Figure 1.5: The flow taken during the throwing of an exception in C++ [33].

1.4.3 Exception Handling in LLVM

The LLVM framework provides the following three IR instructions to handle exceptions [27]. They cooperate closely with the runtime library.

1. `invoke` instruction. It performs a call to a function (like a `call`), but provides an unwind edge for exception propagation.

2. `landingpad` instruction. It is the first instruction of the basic block taken if `invoke` ends up with an exception. It contains the exception handler code to execute and encodes the type of handler, i.e., if it is a call to a destructor (*cleanup*) or if it is part of a `catch` clause (in this case, the exception type is specified).

3. `resume` instruction. It propagates an exception across multiple handlers (for instance, consider nested `catch`-block). It translates into a call to `_Unwind_Resume`, hence it returns control to the personality routine.

Quite unsurprisingly, in light of the above, there is no LLVM instruction for throwing an exception, since its management is entirely left to the runtime library, assisted by the unwind library.

```
define i32 @main(i32 %0, i8** %1) personality i32 (...)*
    @__gxx_personality_v0 {
  %3 = invoke i32 @divide(i32 %0, i32 0)
          to label %12 unwind label %4

4:                                                ; preds = %2
  %5 = landingpad { i8*, i32 } cleanup
          catch i8* @runtime_error
  %6 = extractvalue { i8*, i32 } %5, 1
  %7 = call i32 @llvm.eh.typeid.for(i8* @runtime_error)
  %8 = icmp eq i32 %6, %7
  br i1 %8, label %9, label %14

9:                                                ; preds = %4
  %10 = extractvalue { i8*, i32 } %5, 0
  %11 = call i8* @__cxa_begin_catch(i8* %10)
  br label %12

12:                                               ; preds = %2, %9
  %13 = phi i32 [ 0, %9 ], [ %3, %2 ]
  ret i32 %13

14:                                               ; preds = %4
  resume { i8*, i32 } %5
}
```

Listing 1.4: Example of exceptions handling in LLVM IR. This simplified snippet is the LLVM IR equivalent of the function `main` in Listing 1.3. Since `divide` may throw, it is called through an `invoke` instruction. If an exception is thrown, the unwind edge (`%4`) is taken, whose first instruction is a `landingpad`. If the exception type thrown matches (i.e., it is a `std::runtime_error`), the exception is caught, otherwise the control is returned to the unwind library.

Chapter 2

State Of The Art

This chapter aims to give a detailed insight into the state of the art in coverage-guided binary fuzzing, including AFL in QEMU-*mode*, Honggfuzz + QBDI, Jackalope, QASan and Retrowrite, in order to compare with `rev.ng-fuzz` afterwards.

2.1 AFL in QEMU-*mode*

American Fuzzy Lop (AFL) is a coverage-guided fuzzer written by Michał Zalewski. While it suffices to recompile the program to add coverage instrumentation code when source-code is available, instrumenting binary-only targets need to be accomplished differently. It is possible to leverage dynamic binary rewriting, e.g., via QEMU user-mode emulation (see Section 1.2) [42]. Indeed, as shown in Listing 2.1, AFL extends QEMU by interposing after a TCG basic block has been generated to call a routine that traces the visited edge in the AFL bitmap, stored in a shared memory between QEMU and AFL. To improve speed, newly-translated basic blocks are cached by the fork-server parent for children future use.

The coverage instrumentation is slightly different from the compiler-aided one (see Section 1.3): the current TCG basic block is checked to be within the text section, and it gets scrambled to hide the effects of instruction alignment, given that QEMU maps at fixed location, as shown below.

```
if (block_addr > elf_text_start && block_addr < elf_text_end) {
  cur_location = (block_addr >> 4) ^ (block_addr << 8);
  shared_mem[cur_location ^ prev_location]++;
  prev_location = cur_location >> 1;
}
```

The main drawback of QEMU-*mode* is to incur performance penalties due to expensive instrumentation overhead; indeed, it is roughly 2-5x slower than compile-time instrumentation. On top of this, note that, in order to avert slow execution, when possible, QEMU patches the end of current translated block to directly chain to the next translated block to execute, thus avoiding to go back to the QEMU main emulator loop. However, this prevents AFL from interposing since `cpu_tb_-`

exec (see Listing 2.1) would not be executed. As a result, AFL disables block chaining, though this comes at the further expense of performance.

While there have been attempts to accelerate AFL+QEMU by injecting the instrumentation at translation time, and thus re-enabling block chaining [5] reaching a 3x speedup; AFL did not undergo any remarkable update in a long while. By contrast, a more recent community-driven fork emerged, known as afl++, which has been significantly improving AFL since its release [20]. Some of the enhancements for QEMU-*mode* include in-process mode, deferred forkserver and sanitization.

```
static inline tcg_target_ulong cpu_tb_exec(CPUState *cpu,
    TranslationBlock *itb) {
  CPUArchState *env = cpu->env_ptr;
  uint8_t *tb_ptr = itb->tc_ptr;

  // Record visited TB into AFL bitmap
  afl_maybe_log(itb->pc);

  // Entrypoint into the translated code to be executed
  uintptr_t ret = tcg_qemu_tb_exec(env, tb_ptr);
  // [...]
  return ret;
}
```

Listing 2.1: A simplified extract of the patch to QEMU for AFL. Before starting executing a translated block, the function that yields the coverage map to AFL is called. At program start up, AFL initializes the forkserver as well.

2.2 Honggfuzz & QBDI

Honggfuzz is a evolutionary-based coverage-guided fuzzer with a multi-process and multi-threaded architecture, originally written by Robert Święcki and currently maintained by Google [21]. It is out-of-process, although, similar to libFuzzer, it supports in-process (persistent) mode as well. It starts with a simple corpus of inputs, and expands it through feedback-based coverage metrics. When coupled with QBDI, a dynamic binary instrumentation framework developed by Quarkslab [34], Honggfuzz can perform blackbox binary fuzzing for multiple architectures, by leveraging the coverage instrumentation that can be injected through QBDI [25]. Indeed, QBDI provides API to dynamically interpose on program entry point as well as to trigger an event whenever the execution enters (or exits) a basic block. QBDI exploits such callbacks to simulate the Honggfuzz static instrumentation: a bitmap of shared memory between the monitoring and target process is updated accordingly when a basic block is reached.

While, as said, Honggfuzz provides a way to test new data within the same process (persistent mode), and thus avoiding that QBDI re-writes the program every time, dynamic binary instrumentation comes with a considerable performance cost, which may lead to low fuzzing rates, averaging around 130 exec/s.

2.3 Jackalope

Jackalope is a binary fuzzing library for Windows and macOS based on dynamic instrumentation, written by Ivan Fratric of Google Project Zero [43]. It relies on a dedicated library to add the instrumentation (TinyInst) and has a separate library (LiteCov) to collect the coverage [32]. Jackalope can be used to instrument and analyze different portions of the program, and allows to record the code coverage at different levels (basic blocks, edges and compare instructions). It currently supports generic mutators, but it can be customized to add more complex ones.

The key idea consists in having a debugger that attaches to the original program, keeps monitoring it and gets triggered whenever a general event (e.g., instrumented basic block hit) occurs within the target process. Specifically, after that the relevant module of the target process has been loaded into memory, its text segment is marked as non-executable and the whole area is copied onto the Jackalope address space as well, so that it can be accessed faster. As soon as the original process attempts to execute the target module, an exception is raised and caught by Jackalope, which promptly starts rewriting and instrumenting the hit basic block and all its successors (direct branches and calls are recursively followed). Once done, the instrumented code is copied back into a newly allocated memory region in the target address space; the exception thread instruction pointer is set to point to such area, and the execution can finally resume.

For example, on macOS, in order to give itself debugging capabilities, Jackalope allocates a mach port, a low-level IPC abstraction, within the target process and registers it as exception port. When an exception event is raised, the monitoring process waits for the exception mach message, queued by the kernel, on that port. Once received, the state of the exception thread can be inspected. In order to actually handle the exception, Jackalope uses a routine exposed by the Mach RPC interface (MIG), which internally calls the delegated function for managing the exception (whose implementation is, however, provided by Jackalope). Coupled with the target task port, it can later read and write the target process memory.

While such an approach has been proved to be effective and works well when the instrumented module is reasonably self-contained, exception events, being expensive, have a significant impact on performance. Further slowdowns may be experienced when too many jumps/calls occur from a non-instrumented module into an instrumented one. This may lead to low fuzzing rates, although it must be noted that the instrumentation can be tuned accordingly, if too slow.

2.4 QASan

QASan is a QEMU extension that supports AddressSanitizer (ASan, more in Section 3.2.5) written by Andrea Fioraldi et al. [19]. It uses QEMU user-mode emulation to launch ASan-instrumented programs and allows them to be tested through off-the-shelf fuzzers (e.g., `afl++`). While enabling compiler instrumentation for ASan is straightforward when source code is available, more work is required with binary-only software. To tackle that, QASan customizes QEMU by instrumenting memory accesses at TCG micro-ops generation time, meaning that, when materializing a TCG load/store instruction, further TCG instructions to validate the corresponding shadow address are emitted, as shown in Listing 2.2. If, during the execution, the shadow byte is different from zero, a crash is triggered and a report is generated.

```
void tcg_gen_qemu_ld_i64(TCGv_i64 val, TCGv addr, TCGArg idx,
    TCGMemOp memop)
{
    // [...]
    // Generate a TCG load from guest memory
    gen_ldst_i64(INDEX_op_qemu_ld_i64, val, addr, memop, idx);

    // Instrument the memory access by computing the corresponding
         shadow address
    switch (memop & MO_SIZE) {
        case MO_64: qasan_gen_load8(addr, idx); break;
        case MO_32: qasan_gen_load4(addr, idx); break;
        case MO_16: qasan_gen_load2(addr, idx); break;
        case MO_8:  qasan_gen_load1(addr, idx); break;
        default: qasan_gen_load8(addr, idx); break;
    }
}
```

Listing 2.2: A short extract of how QASan instrumentation happens when a TCG load instruction is being generated (QEMU code). Depending on the access size to the memory address of the load operation, the appropriate checks to compute its corresponding shadow address are inserted.

In addition to including the ASan runtime library, QASan also comes with a custom implementation of such a library as well, which is preloaded within QEMU at program startup and supplies its own routines for handling the shadow memory. In particular, while QEMU is emulating system calls issued by the guest program, QASan interposes by adding a new system call that dispatches to the proper routine in order to pad heap-allocated objects with red zones, update the shadow memory, etc. For efficiency purposes, on x86 a crafted instruction was added which allows direct dispatcher invocation. Finally, specialized implementations of the main allocation and free primitives are provided too. The main source of overhead may come from the runtime translation and its instrumentation.

2.5 Retrowrite

Retrowrite is a static binary rewriting framework, which provides instrumentation to support AFL and AddressSanitizer, written by Sushant Dinesh et al. [15]. Retrowrite implements reassembleable assembly at its core, namely, it generates an instrumented assembler file starting from the disassembly output of the original binary. This comprises two main steps: a first preprocessing phase, which loads the text and data sections of the input program, disassembles it and recovers a best-effort control-flow graph, using popular existing tools (Capstone and pyelftools). The second stage to take place consists in statically identifying references in the raw disassembly and distinguishing them from scalar values, in order to convert addresses pointing to code and data into assembler labels, thus making the disassembled binary relocatable. Conversely, literals and constants values need to remain unchanged. Such a process is referred to as symbolization and it underpins Retrowrite.

Specifically, Retrowrite attempts to reconstruct the assembler label, previously emitted by the compiler, by relying on i) relocation entries information, ii) addresses calculated relative to the program counter and iii) operands to jumps and calls in the recovered control-flow graph. This way, it is able to reassemble position-independent code. Once a reassembleable assembly file is generated, subsequent analyses run (e.g., function identification and ABI detection) in order to add sanitized fuzzing instrumentation properly.

While reassembleable assembly-based rewriting turns out to be efficient, Retrowrite targets only a specific class of binaries, i.e., x64 non-stripped PIC binaries, thus it is strongly dependent on the settings of the binary and architecture.

2.6 Other works

Other coverage-guided binary fuzzers, e.g. TrapFuzz [44], are moderately inspired by the work done by Stefan Nagy et al. [31], which consists in overwriting the first byte of every undiscovered basic block with a breakpoint instruction. During fuzzing, if a test case triggers the interrupt, the byte corresponding to the hit basic block is marked as coverage-increasing in the bitmap. The address of every trap instruction is maintained in a shadow memory so that, after triggering the interrupt, the corresponding address in the shadow memory is retrieved and the original byte at that location can be restored. As the number of hit basic blocks gradually levels off, and so does the overhead, the approach turns out to be efficient.

Chapter 3

`rev.ng-fuzz`: The Fuzzing Framework

In this chapter, we present `rev.ng-fuzz`, a coverage-guided binary fuzzing framework, based on `rev.ng` and LLVM libFuzzer, which allows to fuzz individual functions of binaries for which the source code is not available. The work extends [18]. The framework is laid on the following goals.

Preserve functional correctness. The semantics of the original program needs to be preserved while adding an invasive change as the introduction of coverage-guided fuzzing to `rev.ng`.

Achieve high-performance code. As much as we do not want to compromise the functional correctness, we want to generate high-performance code in order to let the compiler aggressively optimize the generated LLVM IR and speed up the fuzzing rates.

Enable fuzzing of individual functions. Fuzzing functions is as simple as calling the recovered function within a harness. This is achieved by modifying the IR with the instrumentation to enable the use of libFuzzer and by providing an automatically-generated C header with the prototypes of all the collected functions of the original binary. The analyst should not have a hard time creating the fuzzing function.

Enable hooking of functions. The behaviour of the collected functions can be altered by the analyst, who can override such functions by providing his own implementation of them. This can be useful when dealing with, e.g., network packets or data coming from a socket.

Reuse existing sanitizers. In order to enhance the effectiveness of both fuzzing and bug detection, the framework is tightly integrated with sanitizers: SanCov (SanitizerCoverage), which collects the code coverage information at runtime, and ASan (AddressSanitizer), a memory corruption bugs detector. The instrumentation they provide come in form of LLVM modules.

These milestones have been accomplished by developing a series of LLVM passes that run over the recompilable unit of LLVM IR, in particular:

- An instrumentation pass that allows to return from a specific user-defined entrypoint in the program to the fuzzing context (*FuzzingReturn*).
- A pass that generates a C header containing all the a stub for every lifted function that allows direct invocation of the recovered functions (*Wrapper-Generator*).
- A pass that allows to perform function hooking (*FunctionHooking*).

3.1 Design

In this section we break down the underlying mechanism leveraged by `rev.ng` to employ libFuzzer. The intuition revolves around the fact that during the lifting stage, we have the possibility of making both non-invasive and invasive changes over the LLVM IR module produced by the static binary translator. An example of an invasive change is the introduction of the support for coverage-guided fuzzing. In fact, by injecting the necessary instrumentation, we manage to fuzz single functions of the translated binary, paying a relatively low overhead in terms of performance.

More specifically, libFuzzer – which by default requires the source code and relies on compiler instrumentation – can be combined with the IR module. The output is a standalone binary that keeps feeding new inputs to the translated program, which repeatedly executes the fuzzing guide function. Note that, since the translation happens offline – the original binary is never executed –, and so does the instrumentation, our approach purely works ahead of time (statically). Also note that, we do not inject any assembly code, nor do we work on the disassembled binary; instead, we fully rewrite the program by translating it to an intermediate representation, hence adding reliable instrumentation is not too much of an issue.

This way, it is possible to fuzz programs whose source code is unavailable, starting from user-defined entry points and fuzz isolated regions of interest. Furthermore, running a whole program can be sometimes very hard – if feasible at all. For instance, booting a firmware image is not always manageable. In such situations, fuzzing only delimited portions of code might be a winning strategy.

As an example, consider when, after an initial reverse engineering effort, the analyst identifies a function with an argument (e.g., a buffer) that may be user-controlled, or a function that, supposedly, does user-input parsing. A next reasonable step could be to employ fuzzing in order to discover if crafted inputs may lead the program to reach an invalid state, and under certain conditions, to a software vulnerability. That is where `rev.ng-fuzz` comes into play.

Let us review, at a high level, the workflow stages an analyst needs to go through.

1. Lifting. An input program is fed to `rev.ng` which produces a LLVM IR module that can be recompiled into a working program.

2. Entry points identification. A manual reverse engineering phase is performed to identify the entry points and the functions more interesting to fuzz.

3. Fuzzing target function creation. libFuzzer requires to write the harness function, which receives as arguments the fuzzed input buffer and its size, and feeds that buffer into the code to be tested. The analyst needs to implement it.

4. Compilation of the fuzzer. The fuzzing function is linked against the instrumented IR module. Once linked, the output is recompiled back.

5. Fuzzing. The obtained program can be run until a crash is found.

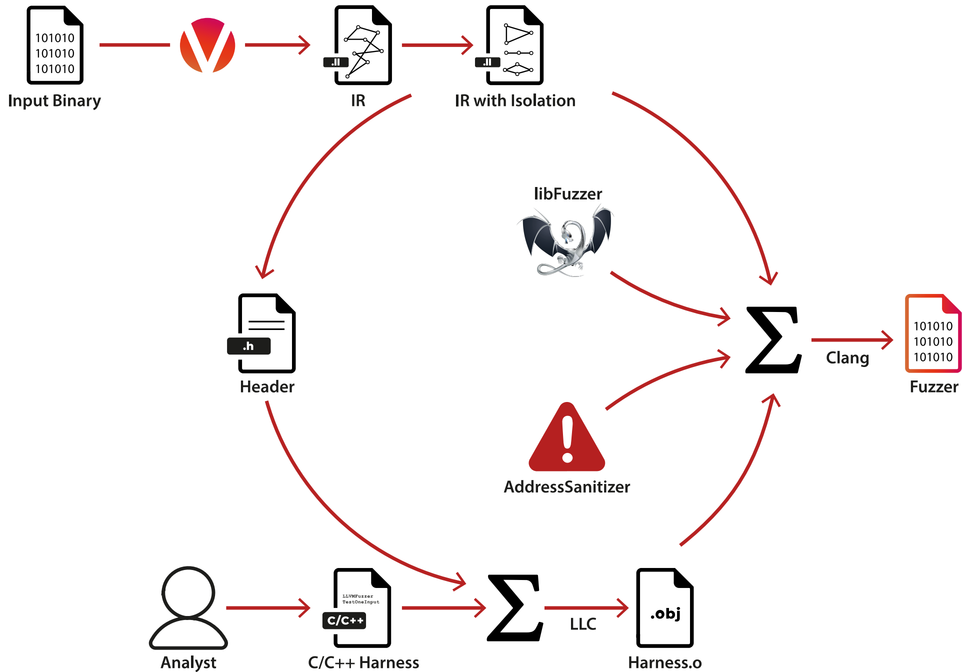


Figure 3.1: An overview of the *fuzzing* process. A binary is given as input to `rev.ng`, which produces an LLVM IR module, containing the recovered functions. After running our optimization passes, `rev.ng-fuzz` produces, among other things, a C header with all the collected functions available for the analyst. Such a header as well as the runtime support library, which contains the harness function written by the analyst, are all compiled and linked together with the fuzz-instrumented LLVM module. Finally, the compiler driver is called to build the fuzzer.

The overall process, illustrated in Figure 3.1, can be summarized as follows.

Input binary → Isolated IR. This is the initial stage, where an input binary goes through the binary translator. After recovering the CFG (by running our own analyses combined with the LLVM ones), a first version of LLVM IR is generated. The IR gets further refined by running a pass that discovers the function boundaries (see Section 1.2.3), where, for each original function found, a new LLVM function is created and added to the module, leading to the so called *Isolated IR*.

Isolated IR → Instrumented IR. The IR is further decorated with the fuzzing instrumentation (more in Section 3.2.2), combined with the sanitizers' one too.

Runtime Support Library + Instrumented IR → Object code. The external runtime support library, which is provided by the analyst and contains, among other things, the initialization function, the harness function dedicated for fuzzing and the inclusion of the header with the prototypes of all the collected isolated functions (so that they can be called from the fuzzing function), is lowered down to object code. The *Instrumented IR* is lowered down to object code as well.

Object code → Fuzzer binary. The various object codes are all linked together into a single file. The sanitizers' libraries are also linked against. On this, the compiler driver is invoked to build the final executable (with the option to enable libFuzzer), and a fully-fledged fuzzer is obtained.

Hence, our approach supports *in-memory coverage-guided fuzzing* of individual functions even when the source code is not available. Since, by design, one can fuzz an ARM binary after translating it to x86-64, it is architecture-independent as well. As aforementioned, one advantage is that we can test distinct portions of code separately, without executing the whole program: every component of the program becomes testable. This is particularly beneficial, for instance, for self-contained libraries and embedded (e.g., baseband) firmware, which, oftentimes, are closed-source.

3.2 Implementation

In this section, we analyze how `rev.ng-fuzz` has been developed in order to achieve the aforementioned goals. We start by illustrating the main idea about the functional correctness and performance that `rev.ng` aims to achieve. Then we examine the fuzzing instrumentation, how we can do function hooking and, finally, how we integrate the sanitizers.

3.2.1 Functional Correctness and Performance

Static binary translation comes with two major challenges, i.e., preserving both the functional correctness and performance of the original program. To have both, we adopt a two-fold approach, which consists in:

- Collecting all the basic blocks of the original program in a single LLVM function (see Section 1.2.1). While not extremely fast and prone to less optimizations, this ensures functional correctness.

- Identifying the start and end addresses of every function and isolating their code into distinct LLVM functions (so called *isolated functions*). This allows for more aggressive optimizations, but may be subject to inaccuracies in the control flow graph recovering phase.

Therefore, we discern between the *isolated realm*, which is composed of the functions previously reconstructed through, as said, a function boundaries detection analysis and the *non-isolated realm*, a single large and semantic-preserving function that contains all the code of the original executable, i.e., `root` (see Section 1.2.1). The latter is intended to start the execution and migrate it to the appropriate isolated function, where it will stay for most of the time. This is particularly desirable considered the fact that the isolated functions have a simpler control flow graph and a smaller scope, whereby the compiler is allowed to generate high-performance code.

Note that, since function boundaries detection and data code separation boil down to be an undecidable problem, we need a way to continue flawlessly with the execution if, for instance, an unexpected jump target is met in an isolated function. Indeed, that would prevent semantic correctness in a one to one execution of the original versus instrumented binary. To address this, `rev.ng` makes use of a mechanism that lets the code run at full speed while inside an isolated function, and fall back to the non-isolated realm only if an unforeseen jump, e.g., a misidentified return instruction, is encountered. The mechanism is based on the *zero-cost exception handling*, which allows to return from any isolated function to `root`, thereby leveraging the stack unwinding process provided by the unwind library.

We are going to dive deeper into these realms and the implementation of such a mechanism in Chapter 4. For now, to start familiarizing ourselves with it, we simply outline how the `root` and the isolated main function (the latter, known as *revng translated block* main) respectively differ from each other.

```
define void @root () {
dispatcher:
  %0 = load i64, i64* @pc
  switch i64 %0 [
    i64 0x1000, label %rtb_main
    i64 0x2000, label %rtb_func1
  ]

rtb_main:
  ; Load rdi and rsi in %1 and %2
  call void @rtb_main(%1, %2)

rtb_func1:
  call void @rtb_func1()
}
```

(a) The `root` function

```
define i64 @rtb_main (i64 %rdi,
    i64 %rsi) {
  call void @rtb_func1()

  call void @rtb_func2()
  ret void
}
```

(b) The isolated `rtb_main function`

Figure 3.2: A comparison between the `root` function and the isolated `main` function. `root` contains all the identified basic blocks of the original binary, and it first tries to dispatch the execution to the proper isolated function. On the left, the main function has been detected among the possible distinct function and is separated from the rest of the code.

3.2.2 Instrumentation

In this section, we analyze the first pass, a LLVM transformation pass, which modifies the LLVM IR adding the fuzzing instrumentation.

Returning to `LLVMFuzzerTestOneInput`

All the functions found during the isolation function process are candidates for fuzzing. Now, we need a way to let the translated program run so that all the initialization routines can be carried out, and pause the execution of the *guest* (the translated program) when the user-defined entrypoint is met (be it the main function or whenever we need to back out of the program). This way, we allow the program to initialize the C runtime environment, including, e.g., the memory allocator, set up static data, prepare the environment variables, etc.

This is what the first instrumentation pass is meant to do, i.e., to allow to return from a specific point of the program chosen by the user (presumably, an isolated function) to `root`. Such a migration is necessary to give control back to `LLVMFuzzerTestOneInput`, the function in charge of guiding the fuzzing process, otherwise the program would simply run until its termination, without fuzzing to be carried out. As an example, suppose we need to initialize the program and return to the fuzzing environment only once a startup routine (`init_vars`) has been executed. The transition back is represented in Figure 3.3.

Figure 3.3: A representation of the transition back to `root` that occurs when initializing fuzzing. The program is let run till, for instance, the end of `rtb_init_vars`, which is an isolated function that is supposed to be executed for starting up the program correctly. Once completed, it returns to the caller, `rtb_main`. Here, instead of continuing the normal execution, the control is transferred back to `root`, which ultimately ends the execution of the *guest* program (leaving it in a valid state) and returns to `LLVMFuzzerTestOneInput`.

As mentioned before, this entails a reverse engineering effort to figure out where it is more suitable to stop this initialization stage and return to the fuzzing context. Once the entrypoint has been found, it is passed as a command line argument to `rev.ng`. In practice, this is an integer that matches the on-disk virtual address of the current executing instruction of the original binary.

The instrumentation has been designed to take advantage of the fact that, every input assembly instruction recovered, when translated to IR, is surrounded by a call to `newpc`, a debugging-aid `rev.ng` function helper that supports program counter tracing. If enabled, a list of all the program counters of the executed instructions is dumped while the translated binary is running. Unsurprisingly, the `newpc` function takes as first argument the program counter:

```
void newpc(uint64_t pc, uint64_t instruction_size, uint32_t
    is_first, uint8_t *vars, ...);
```

This allows us to manipulate the IR in the following way: iterate over all the uses of the program counter of the address provided by the user, and take the use U that matches a call instruction. If the latter is actually a call to the `newpc` function, we found the point in which the instrumentation should be injected.

As shown in Algorithm 3.1 (phase 2), the current basic block is split up immediately after the instruction found. The terminator, namely, a jump to the new basic block is removed, and is replaced with three new IR instructions. The first one stores the value *true* into a sentinel variable, called `NeedToReturnFromFuzzing`, thus signaling `root` that we are returning to it and the migration is to be carried out. The second instruction is a call to `raise_exception_helper`, a `rev.ng` helper function designated to call an external routine that performs the actual migration, based, as said, on the stack unwinding mechanism. Because this function is `noreturn`, the last instruction we emit is an unreachable terminator.

```
Input  : user-defined entrypoint Addr
Output: modified IR with instrumentation
/* Phase 1: find the entrypoint passed by the user as command line
   argument                                                        */
PC = getConstantValue(Addr);
foreach user U in users of PC do
    if U is a CallInst then
        Call = cast⟨CallInst⟩(U);
        F = getCalledFunction(call);
        if F eq newpc then
            FoundInstruction = Call;
        end
    end
end
/* Phase 2: split basic block at entrypoint and emit instructions. */
EntryBB = getParent(FoundInstruction);
SplitBB = splitBasicBlock(nextInstruction(EntryBB));
if SplitBB then
    emit instruction store with NeedToReturnFromFuzzing ← true;
    emit instruction call to raise_exception_helper;
    emit instruction unreachable;
end
```

Algorithm 3.1: The *FuzzingReturn* pass.

```
define dso_local i64 void @rtb_main(i64 %rdi, i64 %rsi) {
%dummy_entry:
  %0 = alloca i64
  store i64 %rdi, i64* @rdi
  store i64 %rsi, i64* @rsi
  br label %rtb_main

rtb_main:                                              ; preds = %dummy_entry

  ; 0x000000000040030e:  push   rbp

  ; mov_i64 tmp0,rbp,
  call void (i64, i64, i32, i8*, ...) @newpc(i64 4195086, i64 1, i32 1, i8* getelementptr
      inbounds ([33 x i8], [33 x i8]* @disam_main, i32 0, i32 0))
  store i1 true, i1* @NeedToReturnFromFuzzing
  call void @raise_exception_helper(i32 0, i64 0, i64 0, i64 0)
  unreachable

__unreachable:                                         ; No predecessors!

  ; mov_i64 tmp0,rbp
  %56 = load i64, i64* @rbp
```

Listing 3.1: Snippet of the non-optimized IR showing the `main` function after running the FuzzingReturn pass. The entrypoint passed by the user was evidently 4195042. Here, immediately after the relative `newpc` call, the original flow of instructions is interrupted and the instrumentation to return to the fuzzing context is injected. The rest of the code within the `main` function is marked as unreachable code, and is going to be removed in subsequent optimization passes.

3.2.3 Generation of the Function Prototypes

The second pass, an analysis one, generates a C header containing wrappers around each isolated function (previously found) (`generated_wrappers.h`), which allow normal function invocation from the `LLVMFuzzerTestOneInput` function, without inspecting the ABI used by the original program. All the generated wrappers start with `srtf_*`, which stands for *safe revng translated functions*.

Let us understand why such wrappers are needed. One could legitimately expect that we simply provide the prototype of the isolated functions (retrieved through a dedicated internal analysis) and then directly call the interested isolated functions for implementing the harness function. However, that could violate the functional correctness of the program. As an example, think of a case in which an indirect jump within an isolated function, has been accidentally identified as a return instruction. That is a clear case in which the execution would leave that function abnormally, thus requiring the intervention of the dispatcher (see Section 1.2.1) inside `root`, from where the execution can resume normally. In other words, if we were to directly call the isolated functions, we would not be able to migrate to the non-isolated realm, in case of jumps to unforeseen basic blocks, since the immediate caller would be the harness function (admittedly, this required a few more iteration of engineering before figuring out that we still need to pass through `root` in order to preserve the correctness of the original binary). The performance is essentially unaffected though. As mentioned, we are going to tackle this dynamic extensively in Chapter 4.

Now, coming to the actual design of these wrappers, recall that `rev.ng`, in order to emulate the state of the CPU, makes use of LLVM global variables for each one of the CPU register or flag (see Section 1.2.2). These are all defined in the LLVM IR module. In the process of automatically generating a header file that declares all the wrappers, there are different CPU variables that are responsible for properly configuring these subroutines – whose purpose, again, is to call the desired isolated function – so that they can be called within the fuzzing function. In fact, we need such wrappers to both respect the functional correctness and to call the intended isolated function. To achieve this, we need to prepare:

1. The emulated stack pointer and program counter. The stack pointer, represented by the CSV `rsp` register, is decremented every time some value needs to be pushed onto the stack and incremented in the opposite case, this way we guarantee it to be always consistent. The program counter, represented by the CSV `pc`, is set up as well, with the on-disk virtual address of the isolated function to be called. Then, `root(rsp)` is called, which takes charge of dispatching the execution to the address of the function set by `pc`. A minor detail: before the actual call, we need to push onto the stack a marker (`0xDEADBEEF`) which just tells that it is time to return when this value is popped off.

2. The function return type. If the corresponding isolated function does not return we simply make the wrapper return `void`. If it is supposed to return a value, we return a `uint64_t`; whereas if it potentially return multiple values, a struct with the detected arguments to be returned is filled.

3. The function formal parameters. The expected function arguments are

passed to the wrappers and correctly sorted according to the translated binary calling convention (x86-64) and are assigned to the CSVs. For example, if the function expects one single formal parameter, the latter is assigned to the CSV corresponding to the `rdi` register. If the function takes a variable number of arguments, all the variadic arguments are traversed and pushed onto the emulated stack, in reverse order (growing downward). This requires adjusting the stack pointer to reserve space for the variadic arguments, and later access it. Note that if the function is believed to be variadic, then the total number of arguments needs to be passed as first parameter. In the future this constraint can be relaxed. An example of wrapper is shown in Listing 3.2.

The quality of the prototypes emitted is as good as the findings another bigger internal analysis (known as *StackAnalysis*), which is queried through a special metadata (`revng.func.entry`) to retrieve the arguments and return values found. At the current stage, we only output `uint64_t` values.

Algorithm 3.2 shows a simplified pseudo code of this instrumentation.

```
Input  : LLVM IR module w/ isolated functions
Output: header file .h declaring the function wrappers
/* Phase 1: query the metadata to see which arguments are alive.   */
create a vector Args of GlobalVariable;
create a vector Ret_values of GlobalVariable;
foreach isolated function F do
    Node = getMetadata("revng.func.entry");
    foreach entry E in Node do
        if E is a argument then
            Args.push(E);
        else
            Ret_values.push(E);
    end
end
/* Phase 2: print the definition of the wrapper and its content.   */
create raw stream object SO, generated_wrappers.h file;
SO ← "#ifndef _GEN_PROTO_H_ #define _GEN_PROTO_H_";
foreach isolated function F do
    printReturnType(SO, F, Ret);
    foreach argument in Args do
        printArg(SO, F, argument);
    end
    printCSVAssignment(SO, F, Args);
    if isVariadic then
        printArgsOnStack(SO, F);
    end
    printCallToRoot(SO, F);
end
SO ← "#endif _GEN_PROTO_H_";
```

Algorithm 3.2: The *WrapperGenerator* pass.

```c
struct unnamed_0 {
  uint64_t field_0;
  uint64_t field_1;
};
struct unnamed_0 srtf_my_func (size_t nargs, uint64_t local_rdi,
    uint64_t local_rsi, uint64_t local_rdx, uint64_t local_rcx,
    uint64_t local_r8, uint64_t local_r9, ...) {
  struct unnamed_0 ret_values;
  target_reg saved_rsp = rsp;

  /* Assign arguments to the CSVs global variable */
  rdi = local_rdi;
  rsi = local_rsi;
  rdx = local_rdx;
  rcx = local_rcx;
  r8  = local_r8;
  r9  = local_r9;

  va_list ap;
  va_start(ap, local_r9);

  /* Push variadic args on the emulated stack */
  rsp -= (nargs * sizeof(target_reg));
  for (int i = 0; i < nargs; ++i)
    *(target_reg*)(rsp + i * sizeof(target_reg)) = va_arg(ap,
        target_reg);
        va_end(ap);

  /* Set up the program counter */
  pc = 0x43DCF8;

  rsp -= sizeof(target_reg);
  *(target_reg*)(uintptr_t)rsp = 0xDEADBEEF;
  root(rsp);
  rsp += (nargs * sizeof(target_reg));

  /* Verify that rsp is consistent after reclaiming space */
  assert(rsp == saved_rsp);
  ret_values = (struct unnamed_0){ .field_0 = rax, .field_1 = rdx
      };
  return ret_values;
}
```

Listing 3.2: Example of an automatically generated wrapper, `srtf_my_func`, around the isolated function `my_func`. First, the local arguments are assigned to the CSVs global variables. Since the function is variadic, the remaining arguments are pushed onto the emulated stack. Then the CSV of the program counter is set to `0x43DCF8`, i.e., the virtual address of the entrypoint of `my_func` and the call to `root` is emitted. Finally, the detected return values are embedded in the struct `unnamed_0`, which is returned.

3.2.4 Hooking

In this section, we examine the third LLVM pass, which allows the user to easily override existing functions (like `malloc` and `free`, `send` and `recv`, and so forth) with own custom implementation. This turns out to be useful when we want to add user-defined callbacks, inject raw data, e.g., when a function expects data coming from a socket or, likewise, intercept a function call for debugging and extending functionalities. To enable the user to perform function hooking, for every isolated function found we take the following steps:

1. Make its symbol have weak linkage. The isolated function (which has been previously recreated with the addition of function arguments and return values in its definition) is changed its linkage and the *weak* attribute is added.

2. Clone the function. The isolated function's formal parameters are mapped to the new function's formal parameters. Then, a LLVM routine (`CloneFunctionInto`) is called, which, among other things, takes as input a map that records the mapping from values in the source function to the values in the cloned one, and returns a copy of the current function. The latter is added it to the IR module.

3. Wrap up the original implementation in a dedicated function. The new function clone is renamed so that it ends with `__impl`: this is going to be the full implementation of the function. The original function body, by contrast, is replaced with a pair of call instruction to the implementation and a return. This way, the original function effectively turns out to be a wrapper around the implementation function.

This altogether allows to circumvent the *One Definition Rule*, which says that in any translation unit, a type, function, or object cannot have more than one definition. Hence, if a symbol is redefined, it will override the original one, but we are still able to call the original function. We can hook a function by adding its redefinition in the runtime support library, which is linked with the IR module. For example, consider we want to hook the libc `malloc` function, linked against the orginal binary. Once the lifting process has completed, `malloc` will have been correctly isolated as `rtb_malloc` in the LLVM IR module. Afterwards, when the hooking pass has finished running, the definition of `rtb_malloc` will look as follows:

```
define weak i8* @rtb_malloc(i64 %rdi) {
  %0 = call i8* @rtb_malloc__impl(i64 %rdi)
  ret i8* %0
}
```

At this point, it suffices to define our own customize version of `rtb_malloc` in the runtime support library, as shown in Listing 3.3, so as to override the `rtb_-malloc` contained in the IR module, which will be removed by the linker since unused, thus obtaining full interposition.

Note that, after our last revision of this pass, the user is now required to pass the name of the functions to be hooked, as argument to the LLVM pass – instead of iterating over all the isolated functions. This was primarily introduced to relieve the LLVM optimization stages so that it should take less time to execute the pass. Since the optimizer may be called again after linking against the support runtime

library (therefore when all the symbols have been resolved), the original functions would have likely not survived the compiler optimizations anyways: the call site would have been directly replaced with a call to the implementation function; so the code size of the IR module would have been roughly the same.

```c
extern void *rtb_malloc__impl(uint64_t rdi);
void *rtb_malloc(uint64_t rdi) {
  printf("Bytes to be allocated: 0x%llx", rdi);

  /* Return the original implementation */
  return rtb_malloc__impl(rdi);
}
```

Listing 3.3: Redefinition of the `rtb_malloc` function. As it can be seen, `rtb_malloc` is hooked with the purpose of logging the number of bytes about to be allocated. Then, the original `malloc` implementation is called.

Algorithm 3.3 shows more thoroughly the aforementioned steps to achieve function hooking.

Input : LLVM IR module
Output: modified IR with hooking instrumentation

```
/* Phase 1: Remap parameters and clone the function */
foreach isolated function F do
    RetType = getType(F);
    NewF = createFunction(RetType, getName(F) + "__impl");
    create a map VMap of ValueToValueMapTy;
    ActualArg = NewF.arg_begin();
    foreach argument I of function F do
        VMap[I] = ActualArg++;
    end
    cloneFunctionInto(NewF, F, VMap);
    /* Phase 2: split basic block at entrypoint of original function
       and emit a call/ret. */
    EntryBB = getParent(F);
    SplitBB = splitBasicBlock(nextInstruction(EntryBB));
    if SplitBB then
        emit instruction call to NewF;
        emit instruction ret with return type RetType;
    end
end
```

Algorithm 3.3: The *FunctionHooking* pass.

3.2.5 Sanitizers

In this section, we see how we have been able to successfully combine fuzzing with sanitization by leveraging two transformation passes that LLVM ships with. This increases the effectiveness of the testing process considerably. The first pass is SanitizerCoverage (SanCov), which monitors the paths that are being exercized by inserting callbacks that implement coverage tracking at different levels (edge coverage in first place) [3]. This is what actually makes the fuzzing *coverage-guided*, since it helps understand which parts of the program were not reached during fuzzing, by differentiating between branches of an `if...else` expression; thus guiding the corpus expansion and mutations as coverage-increasing paths are taken.

To enable SanitizerCoverage, the LLVM optimizer is called with flags which i) decorate the IR with calls for coverage information (`sancov`), and ii) instruct the compiler to record the coverage across initial basic blocks and critical edges (i.e., those edges whose starting and ending blocks have several forward/backward edge, `sanitizer-coverage-level=3`). The actual call inserted at the level chosen, as shown in Figure 3.4, implements simple coverage reporting and visualization for subsequent analyses. We use the one provided by LLVM, but the user can customize the callback according to the needs.

```
rtb_main.0xc:
  call void @__sanitizer_cov_trace_pc()
  %1 = load i64, i64* @rdi
  add i64 %1, 20
  br label %rtb_main.0x20

rtb_main.0x20:                    ; preds = %rtb_main.0xc
  %2 = load i64, i64* @rdi
  store i64 %2, i64* @rbx
  br label %rtb_main.0x2f_crit_edge

rtb_main.0x2f_crit_edge:          ; preds = %rtb_main.0x20
  call void @__sanitizer_cov_trace_pc()
  br label %3
```

Figure 3.4: SanCov annotates the IR with callbacks to provide edge-coverage. Note that some of the edges/blocks may be left uninstrumented, i.e., pruned, if the instrumentation is determined to be redundant.

The other pass is AddressSanitizer (ASan), possibly the most interesting sanitizer for fuzzing, which provides a sound way to detect memory corruptions bugs [37]. It works by instrumenting every load/store instructions, and surrounding dynamic memory allocation, globals and stack objects with extra inaccessible memory (*red zones*) in order to check for possible out-of-bounds accesses. To maintain the state of the application memory, AddressSanitizer uses a *shadow memory*, a separate region of memory (1/8 of the application virtual address space, `mmap`'d at program), which records whether each byte of allocated memory is safe to access or has

been marked as unaddressable (poisoned). Every 8 bytes of application memory are mapped into 1 byte of shadow memory; that is, for each 8 bytes of normal memory, the corresponding byte of the shadow memory is 0 if the memory is legal to access and has another value otherwise. Whenever a load/store occurs, ASan translates a memory address to its corresponding shadow address and the shadow memory is checked. Specifically, when instrumenting an 8-byte memory access, ASan computes the address of the shadow byte as $ShadowAddr = (Addr >> 3) + Offset$, loads $ShadowAddr$ and checks if it is zero. If so, a crash with an error report is obtained. A slightly different computation is applied when instrumenting a 1-, 2-, or 4-byte access. Temporal bugs, like use-after-free issues, are checked as well by restricting the use of freed allocations, so that these regions cannot be immediately reclaimed. Note that, in our case, stack-based objects are not poisoned since it is both non-trivial to achieve and a return address overwrite would be likely caught anyways. Below, Figure 3.5 shows how the shadow memory looks like.

Figure 3.5: AddressSanitizer: a representation of the shadow memory [6]. Poisoned redzones are created around each each dynamically allocated object. If an application tries to read/write an object out of its limits, a crash is triggered.

To enable AddressSanitizer, the LLVM optimizer is called with flags `asan`, `asan-module`, `asan-instrument-writes` and `asan-instrument-reads`, which transform the IR module generated by `rev.ng` with the ASan instrumentation. Note that, the load/store instructions are instrumented only on functions marked with the attribute `Attribute::SanitizeAddress`, thus we iterate over each isolated function found and annotate it with such an attribute. The other component that AddressSanitizer comes with is `libasan.a`, the runtime library, whose purpose is to initialize the shadow memory, create the red zones around objects, replace `malloc`/`free` with specialized implementation and do error reporting. To include the library, it suffices to link against during recompilation of the IR module. Then, in the external support file, we manually replace the default `malloc` and `free` with the ASan replacement ones, through function hooking. This is illustrated in Listing 3.4. Note that, unlike the previous hooking example, this time, since we are not returning to a LLVM function, we manually need to pop the emulated program counter off the stack in order to return to the caller – while continuing to return the value. This is embedded in the `EMULATED_RET` macro.

```
#define EMULATED_RET do { \
  pc = *(target_reg*)(uintptr_t)rsp; \
  rsp += sizeof(target_reg); \
} while(0)

extern void *__interceptor_malloc(size_t size);
void *rtb_malloc(uint64_t rdi) {
  void *ptr = __interceptor_malloc(rdi);
  EMULATED_RET;
  return ptr;
}

extern void __interceptor_free(void *ptr);
void rtb_free(uint64_t rdi) {
  __interceptor_free((void*)rdi);
  EMULATED_RET;
}
```

Listing 3.4: Redefinition of the isolated `malloc` function (`rtb_malloc`) with a call to the ASan `malloc` replacement in the support file. This time, since we call a function out of the IR module, we manually need to set the emulated program counter with the value pointed by the stack pointer to return to the caller appropriately.

```
rtb_main.0xc:
  call i64 @rtb_foo()
  ; Check and jump depending on ret value @rax saved in %0
  br i1 %0, label %rtb_main.0x20, label %rtb_bad_return_pc

rtb_main.0x20:                 ; preds = %rtb_main.0xc
  %1 = load i64, i64* @rax
  ; Computer val = (addr >> 3) + offset
  %2 = inttoptr i64 %1 to i8*
  %3 = ptrtoint i8* %2 to i64
  %4 = lshr i64 %3, 3
  %5 = add i64 %4, 2147450880
  %6 = inttoptr i64 %5 to i8*
  %7 = load i8, i8* %6
  %8 = icmp ne i8 %7, 0
  ; Check if (*val != 0) goto %8
  br i1 %8, label %9, label %10

%9:                            ; preds = %rtb_main.0x20
  call void @__asan_report_store1(i64 %2)
  unreachable
```

Figure 3.6: The AddressSanitizer instrumentation. The memory access of the `rax` CSV is instrumented, i.e., the corresponding shadow address is computed, and the error function `__asan_report_store` is called, if `rax` is accessed out of its bounds.

3.3 Execution Flow

In this section, we analyze the overall execution flow. It begins with the invocation of the fuzzing target, which is linked together with the LLVM IR module. This function is repeatedly called with the mutated input buffer generated by libFuzzer. Within it, an initialization routine is called the very first time, shown in Listing 3.5, with the aim of:

1. Reserving a portion of memory via that is meant to emulate the *stack* of the original program. The returned pointer, which evidently will be used as stack pointer, is adjusted to point at the end of the area, minus 0x1000 bytes for safety reasons. This way, we can reproduce and be congruous with the behaviour of the original stack, which in the x86-64 ABI grows downward, as illustrated in Figure 3.7. Then, the memory area is assigned to `rsp`, the CSV that corresponds to the emulated stack pointer. It will be passed to the `root` function as argument. A memory region to emulate the *heap* is allocated as well. This is passed to a QEMU helper routine (`target_set_brk`), which sets some variables later used by the implementation of the emulated `brk` syscall.

2. Initializing some other mechanics, including the data structures for `ioctl` handling (via `syscall_init` QEMU helper) as well as a custom SIGSEGV handler.

```
void InitializeFuzzing(void) {
  // Allocate and initialize the stack.
  void *stack = mmap(NULL,
                     16 * 0x100000,
                     PROT_READ | PROT_WRITE,
                     MAP_ANONYMOUS | MAP_32BIT | MAP_PRIVATE,
                     -1,
                     0) + 16 * 0x100000 - 0x1000;
  assert(stack != MAP_FAILED);

  // Set rsp to the stack base.
  rsp = (target_reg)stack;

  // Allocate the brk page as well.
  void *brk = mmap(...)
  target_set_brk(brk);

  // Initialize the syscall system and SIGSEGV handler.
  syscall_init();
  install_sigsegv_handler();

  // Run the program till the user-defined entrypoint is met.
  pc = Address_start;
  root(rsp);
}
```

Listing 3.5: An abridged version of the `InitializeFuzzing` routine.

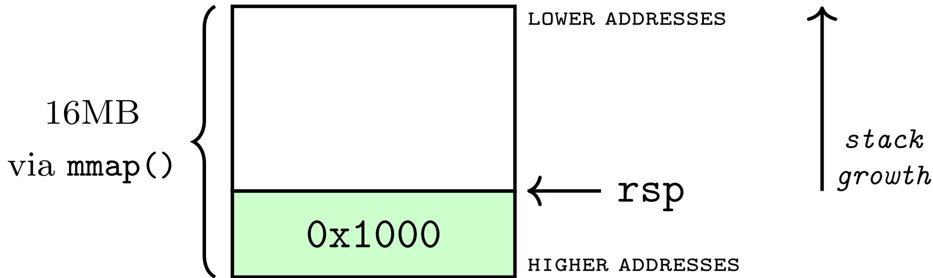


Figure 3.7: A representation of the memory region for the emulated stack.

Therefore, when the translated binary, namely, the fuzzer, starts executing, the flow will look as follows: the main function of libFuzzer (`FuzzerDriver`) is called, which, after mutating the corpus, invokes the `LLVMFuzzerTestOneInput` user callback function, shown in Listing 3.6. As said, the coverage instrumentation will continuously provide feedback to the fuzzer. Within this function, our initialization fuzzing routine (`InitializeFuzzing`) gets called. Note that, libFuzzer allows us to provide an init function (`LLVMFuzzerInitialize`), however, since we did not need to access `argv`/`argc`, we implemented a custom initialization routine. Once completed, the control is passed to the `root` function – as shown in Listing 3.5 –, which dispatches the execution to the `_start` routine of the original (*guest*) program. This is going to run until the user-supplied entrypoint is met, whereby the control is transferred back to the `LLVMFuzzerTestOneInput`. From there, all the isolated functions collected are candidates for fuzzing.

```
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  static bool isFuzzingInitialized = 0;
  if (!isFuzzingInitialized) {
    isFuzzingInitialized = 1;
    InitializeFuzzing();
  }

  // Call functions targets here.
  srtf_fuzzMe(data, size);
  return 0;
}
```

Listing 3.6: The `LLVMFuzzerTestOneInput` harness function that the analyst needs to implement. The statically initialized boolean guarantees that `InitializeFuzzing` is called only the first time; then all the functions found of the original program can be potentially fuzzed by calling their wrappers.

3.3.1 Compilation Stage

During the translation process, the LLVM optimizer (*opt*) runs all our transformation and analysis passes on the lifted IR. The runtime support library containing, among other things, the `LLVMFuzzerTestOneInput` routine and the redefinition of the functions to be hooked is also compiled to LLVM IR and linked to the translated code to form a single LLVM IR object, which is further serialized to bitcode, and ultimately lowered back down to object code. For better fuzzing performance, the LLVM static compiler (llc), is called with optimization level `O2`. The object file is passed to the compiler driver (Clang), which is invoked with flag `-fsanitize=address` in order to enable libFuzzer, and the fuzzer binary is built. Eventually, the LLVM optimization pipeline looks as follows:

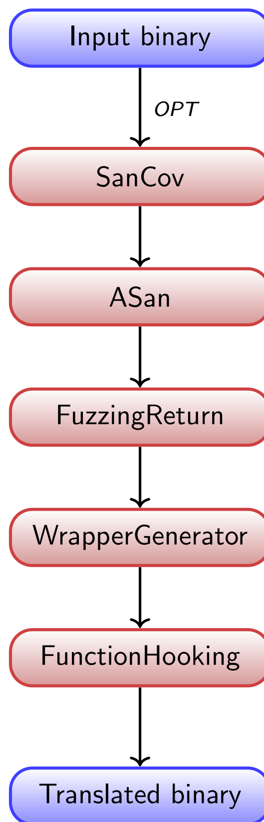


Figure 3.8: The LLVM optimization passes for `rev.ng-fuzz`. At first, the IR decorated with the instrumentation provided by SanCov and ASan. Our passes run next: *FuzzingReturn*, which instruments the IR too for fuzzing purposes, *WrapperGenerator*, which supplies to the analyst an header with the function prototypes and *FunctionHooking*, which, optionally, allows to hook functions.

Chapter 4

Make `EnforceABI` Great Again

In this chapter, we are going to show how we ameliorated `EnforceABI`, a `rev.ng` transformation pass, that promotes the CSVs (variables representing the CPU state, see Section 1.2.2) from being LLVM global variables to local variables and formal parameters of a function, according to the architecture-agnostic ABI analysis. The goal is to squeeze out as much performance as possible by generating an enriched LLVM IR as output of the static binary translator, whereby further compiler optimizations can take place – yet still being semantics-preserving, i.e., without violating the functional correctness of the original program.

We start off by introducing the separation of the translated code between the faster *isolated* realm and the safer *non-isolated* one, then we cover the exception handling mechanism – which allows, at runtime, the transition from the faster realm to the other in case the recovery of the control-flow graph erred – and how we revisited this mechanism so that we can now fully make use of it. Finally, we focus on the enhancements of EnforceABI, which can be fully leveraged by the static binary translator, i.e., `rev.ng` can generate a binary optimized with EnforceABI and, now, execute it.

While our main motivation was to make `rev.ng-fuzz` benefit from this, i.e., with the purpose of speeding up the fuzzing rates; in a more long-term view, this work can be construed as an improvement over *Performance, Correctness, Exceptions: Pick Three*, formerly presented at NDSS Symposium '19 [22], and ultimately aims at making the execution of the translated binary faster, since more optimization opportunities can be taken when the compiler deals with local variables, rather than global ones.

4.1 Isolation of Functions

To get acquainted with our contribution, this section begins by illustrating the two realms that exist in `rev.ng`, the isolated and non-isolated one, which was briefly touched upon in Section 3.2.1, and why this segregation is beneficial. Then, we introduce *Function Isolation*, an existing `rev.ng` transformation pass, that actually creates LLVM functions for the recovered functions of the original program.

Recall from Section 1.2.3 that in stripped or optimized binaries, detecting the boundaries of functions is an undecidable problem: oftentimes, they get blurry due to compiler optimizations to an extent that the function prologues and epilogues may not follow standard patterns, or may be missing altogether. Consider the tail-call optimization: if it was determined that a routine calls another function as the very last thing, the former function may not end with a return, but directly with a jump to the other function, preventing the allocation of a new stack frame. As a result, the more optimized the binary, the harder the function detection problem. Likewise, since we are dealing with static analysis, a meaningful reconstruction of the control-flow graph of a program is challenging. Resolving the branch target for indirect calls and indirect jumps requires estimating the content of pointer variables, namely, both the register value and the memory location. While there exist points-to analyses that can help disambiguate indirect accesses by essentially propagating pointer assignments across program data flows according to, e.g., the type; the concrete behaviour of indirect branches is known only at runtime. Hence, pointer analyses notwithstanding, the original call-graph cannot be fully reconstructed either way. That said though, the more accurate the reconstruction of the control-flow graph, the better the function boundaries detection.

The Function Isolation pass. To overcome these issues, `rev.ng` was designed to adopt a hybrid approach: all the recovered code from the original executable is contained in a single large function, known as `root` (non-isolated realm). Subsequently, a Function Boundary Detection Analysis (FBDA) pass attempts to discover the start and end addresses of the functions of the original binary, by labeling all the basic blocks that belong to the same function. These information are supplied to Function Isolation, the pass in charge of collecting and cloning all these basic blocks into a new LLVM function, along with some minor refinement (e.g., an ending basic block in `root` never returns, instead jumps to the dispatcher, whereas a normal function terminates with a LLVM `ret` instruction, hence, the instruction change).

Each function previously identified is disentangled from the rest of the code to form a so called *isolated function*. The set of these functions forms the isolated realm. At program launch, the `root`'s dispatcher (see Section 1.2.1) diverts the execution to the proper basic block that will call its isolated function, according to the value of `pc` (the global variable emulating the program counter).

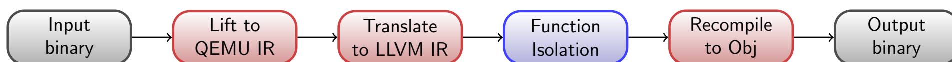


Figure 4.1: The `rev.ng` pipeline with Function Isolation. The whole isolation process really comes down to dividing the code in two separate portions: the isolated realm, namely all the recovered functions of the original program in form of distinct LLVM functions, and the non-isolated realm, that is the `root` function. The new generated IR is known as *Isolated IR*.

This way, the control-flow graph of the isolated functions is much cleaner and allows for more compiler optimizations, but may be incomplete and have missing edges, since the jump targets of indirect calls is not always known a priori, as

mentioned before. Hence the need for making these two realms coexist with one another. In fact, while it is desirable that the program runs over the isolated realm for most of the time (being high-performance code), the execution may unexpectedly leave the isolated realm and continue in the non-isolated one from time to time. This could occur primarily in two cases: i) with indirect branches whose target cannot be determined statically and ii) when the return instruction has been misidentified and does not correspond to the expected fall-through address of the function call, i.e., the value that is placed into the emulated program counter is not correct; after all, a return instruction is simply just an indirect jump.

Particularly, all the disassembled indirect jumps are treated the following way in the IR: a call to a dedicated function (`function_dispatcher`) is emitted, which evaluates at runtime the program counter and jumps to the correct basic block, i.e., the target of the indirect branch. It works the same way as the dispatcher in `root` but only contains the basic blocks corresponding to function entry points. Should the target still be unresolved, the execution would switch to the non-isolated realm, i.e., to `root`, which has all the possible basic blocks and can determine dynamically the actual target to jump to. Consequently, `root` is executed only when an isolated function performs a jump to an unpredicted location, thus effectively acting as a fail-safe solution.

As we are going to see in Section 4.1.1, the migration from an isolated function to `root` is carried out using the Itanium C++ ABI *zero-cost exception handling*, i.e., the mechanism through which C++ achieves the stack unwinding when throwing an exception via the `throw` instruction; the latter is then translated by the compiler into a call to the `__cxa_throw` runtime library routine.

The transition from the isolated realm to the non-isolated one is exemplified in Figure 4.2.

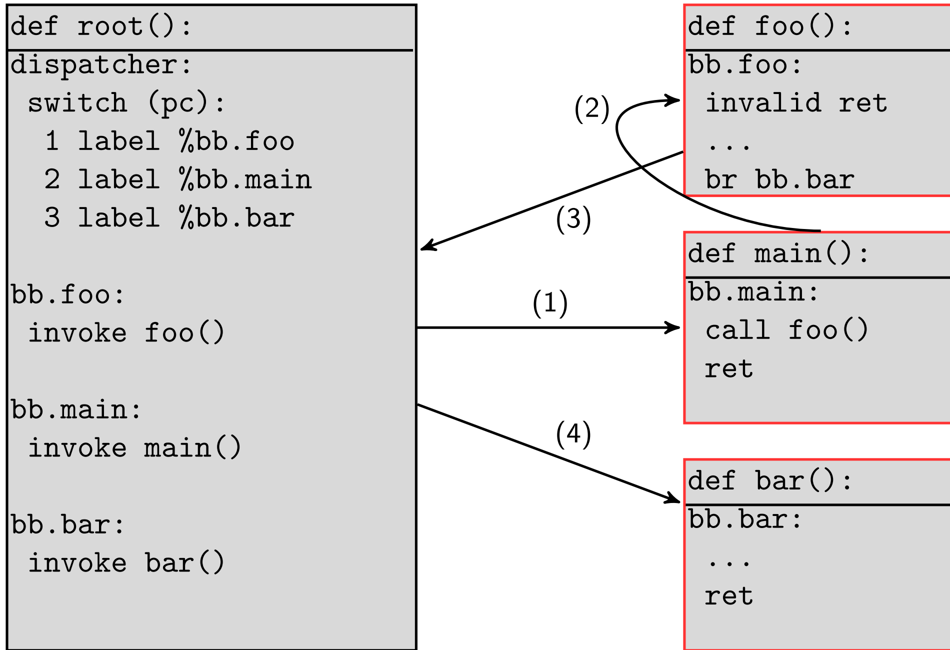


Figure 4.2: A representation of the transition from an isolated function to `root` and viceversa. Suppose that the flow of the original program consisted of `main` → `foo` → `bar`, where the latter is indirectly called by `foo`. `rev.ng` is expected to properly recover such a flow. However, say for instance that, while detecting the function boundaries, the indirect jump (tail call optimized into a jump) in `foo` has been accidentally identified as a return instruction. With the design of Function Isolation in mind, the translated execution looks as follows: `root` dispatches the execution to the basic block of `main` identified as a function entry, which calls its isolated function (1). From there, `main` calls the isolated function `foo` (2). Here, the incongruous situation is detected and and exception is thrown; the stack frames allocated so far (namely, the ones of `main` and `foo`) are unwound up, and the control is passed back to `root` (3). By matching the correct fall-through address that `foo` was supposed to jump to, `root` re-dispatches again the execution, this time to the entry basic block of `bar`, where finally its isolated function can be called (4).

4.1.1 Exception Handling in `rev.ng`

This section analyzes how `rev.ng` exploits the exception handling mechanism (see Section 1.4), with a focus on our improvements. Since exception handling is notoriously complex, the ability to reuse this mechanism relieved `rev.ng` by not reinventing the wheel, and enabled to implement an architecture-agnostic migration mechanism through off-the-shelf components.

Recall from Section 1.4.3 that in LLVM IR, when calling a function, the `invoke` instruction may throw (unlike the simpler `call` one, which does not). It takes two basic blocks as possible branches once the function called has returned: the first one, for continuing the normal flow, and the second one, which is taken if an exception has been raised or is being propagated from the callee. This latter basic block needs to start with the `landingpad` instruction, which encodes the type of landing block (e.g., catch block, C++ destructor, etc.).

As aforementioned, when an isolated function jumps to an unexpected basic block, the execution flow may be redirected to `root`, which eventually dispatches the execution to the proper basic block. To achieve this transfer of control, the faulty isolated function calls an internal `rev.ng` helper, `raise_exception_helper`, that creates an *exception object* and passes it along to the unwind library, by calling the platform-specific `_Unwind_RaiseException` routine, part of `libunwind` in LLVM and described by the Itanium ABI interface on Intel platforms.

```
define void @root() {
dispatcher:
  ; ...

bb.foo:
  invoke void @bb.foo() to
    label %normal
    unwind label %catchblock

normal:
  ; ...

catchblock:
  landingpad %0
    catch i8* null
  br label %dispatcher
}
```

```
define void @bb.foo() {
  bb.foo:
    ; ...

  unexpectedpc:
    call void
      @raise_exception_helper()
    unreachable
}
```

Figure 4.3: On the left, it is represented `root` with a basic block, `bb.foo`, that will call its isolated function to achieve more performance, through the `invoke` instruction. On the right, the corresponding isolated function. If an indirect jump within the isolated function `foo` is met, the correct execution could supposedly blow up. When this occurs, the basic block `unexpectedpc` is immediately taken and an exception is thrown. Eventually, it will be caught by the basic block `catch` in `root`.

From there, the control is left entirely to the unwind library. More precisely, the decision on which exception handler to execute is deferred to the *personality routine* (see Section 1.4.1), which consults the compiler-generated unwind tables to find out pertinent information about the handlers. Once the personality routine successfully has found the exception handler, the control is transferred to this handler so that it can execute, and eventually the exception will either resume propagation or end (this happens when the `catch`-block in `root` is reached).

To allow the execution of a binary optimized with EnforceABI, as anticipated, the exception handling mechanism has been revisited. Prior to our contribution, `rev.ng` featured a limited support to execute the `catch`-block handlers while unwinding the activation record of each called function, due to lack of implementation of the personality routine. Particularly, during the search phase of a handler, the `rev.ng` custom personality function was not actually identifying the address of the exception handler specific for that function that should have been unwound (it was simply returning `_URC_HANDLER_FOUND` code without performing the actual identification). Likewise, in the next phase, the control was not transferred to the exception handler, despite being reported as found (`_URC_INSTALL_CONTEXT` code was returned). Therefore, in practice, no exception handler of any function could be executed but the single one in `root`.

This issue was addressed by installing on each isolated function the GNU C++ personality routine `__gxx_personality_v0`. This way, when an exception is thrown, the landing block of each active call frame – if it is provided – can be executed, and the exception is guaranteed to properly continue propagating thanks to the `resume` instruction; which is now the `landingpad` terminator that returns control to `libunwind`. After this change, the prototype of an isolated function looks as follows:

```
define i64 @bb.main(i64 %rdi, i64 %rsi) personality i32 (...)*
    @__gxx_personality_v0
```

This also opened up to simplification of the code responsible for catching the exception in `root`, including the removal of a global variable used as a sentinel to signal that an exception was being raised. Listing 4.1 shows the `rev.ng` helper in charge of throwing an exception and transferring the control to the unwind library.

```
noreturn void raise_exception_helper() {
  // Declare the exception object
  static struct _Unwind_Exception exc;

  // Raise the exception using the function provided by the unwind
      library
  _Unwind_RaiseException(&exc);
}
```

Listing 4.1: `raise_exception_helper` is the helper function responsible for creating and throwing the exception and is called whenever the return address does not match with the expected fall-through basic block in an isolated function.

4.2 ABI Enforcement

`EnforceABI` is a `rev.ng` transformation pass that promotes a set of the CSVs to formal parameters as well as to return value of a function. To do so, it takes advantage of the results of a more complex analysis, known as *Stack Analysis*, which builds upon further subanalyses that identify the number and location of arguments and return values in a ABI-agnostic and conservative way (no assumption on specific calling conventions). Briefly, the Stack Analysis follows the evolution of the stack (and in general, the state of the CPU) in order to tell how it is used within a function, by tracking down which registers are being used on the load and store instructions and how they access the slots of the stack. While no hypothesis is done on the underlying architecture, there are some assumptions we do that generalize well across different architectures. For example, explicitly callee-saved registers, i.e., registers that need to be preserved across function calls whose initial value is the same on exit, can never be arguments or return values of a function. Overall, this allows to analyze the usage of the registers; collect as much information as possible and combine it to retrieve the arguments and return values for each function and call site. EnforceABI exploits the information provided by the Stack Analysis, exposed through metadata, and propagates such information to the isolated functions by transforming the CSVs global variables into local variables that get allocated on the stack and are returned to the caller. This leads to a simpler IR to analyze (more abstractions is recovered) as well as further optimizations.

To get an idea, Listing 4.2 shows how the declaration of the isolated function `memcpy` looks like before and after running EnforceABI. As it can be seen, the function prototype changes radically. In fact, without EnforceABI, the function accepts no arguments and does not need to return anything since all the return values are stored in global variables, which have visibility of all the functions within the LLVM IR module. Conversely, after running EnforceABI, the global variables representing the CPU registers no longer exist within the isolated functions: they all have been replaced with local variables that do not persist across functions. More in detail, to run the code translated with EnforceABI, this pass must have at least the following characteristics implemented:

1. Recreate the function prototype. The prototype of each isolated function needs to be rewritten by including the detected function arguments and the return type (namely, distinguishing if the function returns or not).

2. Create local variables for every CSV and create the return values. Besides allocating the other automatic variables, every CSV used within the function is now also stack-allocated (created through an `alloca` instruction), and is either zero-initialized or, if passed as function argument, the newly created CSV is assigned the argument. The same is done for the return values, where all the candidate values are inserted into an aggregate type, which is returned.

3. Replace the CSVs with the local variables. All the uses of the global variables need to be substituted with the dedicated stack-allocated variable.

4. Handle the call site. When calling a function, the caller is responsible for loading the CSVs and passing them as arguments to the call site (the function is called with a `call` instruction) and extracting the return values.

Before EnforceABI

```
define void @bb.memcpy() {
entry:
  %0 = alloca i64, align 8
  br label %bb.memcpy

bb.memcpy:                                    ; preds = %entry
  ; 0x0000000000403c24:  mov    rax,rdi
  %1 = load i64, i64* @rdi, align 8
  store i64 %1, i64* @rax, align 8

  ; ...

  ; 0x0000000000403c55:  ret
  ; Pop the return address off the stack, load it in %70 and
      increment the stack pointer
  store i64 %70, i64* @pc, align 8
  ret void
}
```

After EnforceABI

```
define i8* @bb.memcpy(i64 %rdi, i64 %rsi, i64 %rdx) {
entry:
  %rdi_local = alloca i64, align 8
  %rsi_local = alloca i64, align 8
  %rdx_local = alloca i64, align 8
  %rax_local = alloca i64, align 8
  %0 = alloca i64, align 8
  store i64 %rdi, i64* %rdi_local, align 8
  store i64 %rsi, i64* %rsi_local, align 8
  store i64 %rdx, i64* %rdx_local, align 8
  br label %bb.memcpy

bb.memcpy:                                    ; preds = %entry
  ; 0x0000000000403c24:  mov    rax,rdi
  %1 = load i64, i64* %rdi_local, align 8
  store i64 %1, i64* %rax_local, align 8

  ; ...

  ; 0x0000000000403c55:  ret
  store i64 %72, i64* @pc, align 8
  %73 = load i64, i64* %rax
  %mrv = insertvalue {i64} undef, i64 %73, 0
  ret i64 %mrv
}
```

Listing 4.2: A comparison of the LLVM isolated function `memcpy` before and after running the `EnforceABI` pass. FunctionIsolation discovers and isolates `memcpy`, whereas EnforceABI is concerned with the promotion of the CSVs to function parameters and return values.

4.2.1 Execute `EnforceABI`'d Code

This section delves into our efforts to come up with a working translated binary that has gone through the EnforceABI pass, after being optimized with Function Isolation. The translated binary will contain all the isolated functions recovered of the original program, plus all the CSVs will have been converted to local variables (as shown in Listing 4.2). Note that the promotion to local variables only involves the isolated functions; the CSVs used within the `root` function – which is not meant to be optimized – are left untouched, i.e., they are still global variables. It should be pointed out that EnforceABI was originally devised to be used for decompilation purposes, in the attempt to promote the identified variables to function arguments and return values to reconstruct the function signature and facilitate reverse engineering tasks. Now, our goal is to allow to re-execute the translated program as well, and there are primarily two rationales behind this:

- It enables the compiler to perform more aggressive optimizations when dealing with variables whose scope is narrower, and, in general, this can massively improve the performance as well as reduce the memory traffic. It is way more difficult for the compiler to optimize code that involves global variables, given that they can be referenced from many contexts.

- It offers guarantees of robustness of the decompiled code and validates that the decompiled adheres to the semantic of the original program.

One may wonder whether a binary translated with EnforceABI could be directly run as such, i.e., if it is possible, after lifting a program, to run the EnforceABI transformation over the IR, recompile the module and run the program so that it executes correctly – without further changes.

It turns out that the current design of EnforceABI does not really encompass the re-execution of the translated code, thus, even though Listing 4.2 is semantically correct, it cannot function seamlessly. The reason lies in the fact that, as explained in Section 4.1, there may be unexpected branches within an isolated function that could potentially break the correctness of the program and lead to faults. If the execution happens to go back to `root`, after an exception has been thrown, which state of the CSVs would `root` see? Before running the EnforceABI pass, the CSVs are in the form of global variables, thus, if an isolated function clobbered a CSV, the changes would be immediately visible to `root` too. Now, however, the CSVs turned into automatic variables, and have local scope, hence the CSVs in `root` would not reflect the last changes at all.

When an isolated function throws an exception, the key idea in order to have the execution continue working correctly is to leverage the `catch`-block for serializing the state of the local CSVs, that is, to set the global CSVs with the value of the local CPU variables clobbered by the innermost function among the active stack frames (the one that raises the exception); then have the caller's exception handler to save those CSVs that have not been clobbered yet, and so forth while unwinding the stack, until the handler in `root` is reached, which will jump to the dispatcher. This serialization mechanism that occurs at each handler of the frames to be unwound is illustrated in Figure 4.6. At this point, whichever basic block is going to be called

by the dispatcher in `root`, it will use the global variables CSVs, which will reflect the very last value that was written into them.

Therefore, restoring the value of the local CSVs into their global variable counterpart effectively transforms the landing pad into a *safeguard trampoline*. Moreover, not only does this mechanism allow to not violate the functional correctness of the translated program, but it guarantees that the performance hit is paid only in such exceptional situations.

An overview of the execution flow that occurs when an exception is thrown within an isolated function is shown in Figure 4.5.

Before EnforceABI

```
store i64 4209700, i64* @pc, align 8
call void @bb.memcpy()
%1 = load i64, i64* @pc
%2 = icmp eq i64 %1, 4208437
br i1 %2, label %bb.main.0x20, label %bb.bad_return_pc
```

After EnforceABI w/ `invoke` transformation

```
store i64 4209700, i64* @pc, align 8
%1 = load i64, i64* %rdi_local
%2 = load i64, i64* %rsi_local
%3 = load i64, i64* %rdx_local
%4 = invoke i8* @bb.memcpy(i64 %0, i64 %1, i64 %2)
        to label %bb.continuation unwind label %bb.csv_restore

bb.continuation:
  %5 = extractvalue i8* %4, 0
  store i64 %5, i64* %rax_local
  %6 = load i64, i64* @pc
  %7 = icmp eq i64 %6, 4208437
  br i1 %7, label %bb.main.0x20, label %bb.bad_return_pc
```

Listing 4.3: A comparison of a call site, `memcpy`, before and after EnforceABI with our changes. In addition to loading the content of the memory locations passed as function arguments as well as extracting the return values, now, the `call` instruction has been substituted with an `invoke` in order to support exceptions (as we are going to see soon in Section 4.2.1). Observe the difference: if an exception is thrown within `memcpy`, then the basic block `bb.csv_restore` is taken to serialize the CSVs (illustrated in Figure 4.4). Instead, if upon returning from `memcpy`, the return address has been misidentified but `memcpy` executes successfully, the check against the `pc` CSV fails, thus taking `bb.bad_return_pc`, which raises an exception. After that, `bb.csv_restore` is taken.

Improvements over `EnforceABI`

At the implementation level, EnforceABI has been subject to the following additions and changes.

1. `call` to `invoke` instruction replacement. All the `call` instructions have been transformed to `invoke` ones in order to either let the flow continue normally or branch to an exception handler (when an exception is thrown or being propagated). As a result, this requires to create two additional basic blocks to jump to for each `invoke` added and requires to properly extract the values returned by the function called as well. Listing 4.3 shows how the call site changes among the two different instructions. One may notice that `invoke` instructions, being a terminator, slightly complicate the control-flow graph within a function: while the cost to pay is relatively small, this is the trade-off to be semantics-preserving.

2. Landing pad installation for serializing the CSVs. This is the most remarkable change which the translated code execution relies on. First, a unique landing pad is installed on every isolated function, shown in Figure 4.4, which is the first basic block taken after the unwind library has handled the exception and has returned the control to user code. We maintain a zero-initialized global variable used as a bit-mask to say if a CSV has already been restored or not.

The intuition is to have the faulty function to serialize all the CSVs that it writes a value into, then, have the caller to serialize the remaining CSVs, and so forth. Also, for debugging purposes, in order to figure out whether an isolated function writes into a specific CSV, the CSVs local variables are initialized to a magic value (`0xDEADBEEF`). If at the time of serialization, it is found to have the magic number, then we simply assign the global CSV to itself and move on to the next CSV. This process guarantees that at the end of the stack unwinding, `root` sees the correct values of the CSVs to correctly resume the execution at the point where it erroneously diverged within the isolated function.

Such a serialization mechanism is represented in C by the following snippet for the CSVs of `rdi` and `rax`.

```c
#define RDI (1 << 0x1)
#define RAX (1 << 0x2)

uint32_t RestoreCSVBitMask = 0;
if (!(RestoreCSVBitMask & RDI) && rdi_local != 0xDEADBEEF) {
  g_rdi = local_rdi;
  RestoreCSVBitMask |= RDI;
} else
  g_rdi = g_rdi;

if (!(RestoreCSVBitMask & RAX) && rax_local != 0xDEADBEEF) {
  g_rax = local_rax;
  RestoreCSVBitMask |= RAX;
} else
  g_rax = g_rax;
```

3. Inlining of the function_dispatcher. Recall that when the target of an indirect branch cannot be statically resolved, a call to function_dispatcher is emitted, which contains a jump-table with entries, those basic block that represent the entry point of a function. Now, since we need to pass the arguments to the call site, the indirect calls are handled as follows: the jump-table has been inlined (thus, function_dispatcher no longer exists) and the possible basic blocks to branch include all those that match the same number of function arguments. If the target is still not determined, it is jumped to raise_exception_helper, which throws an exception.

In order to achieve more performance, the optimizer is called over the EnforceABI'd LLVM IR with optimization level O2, before it is recompiled to executable code. Some of the optimizations include SROA, which attempts to eliminate the allocas instruction and promote them to SSA values; InstCombine which combines instructions to form fewer and simpler ones, and CSE, which removes common subexpression computation.

```
bb.csv_restore:
  %lpad = landingpad { i8*, i32 } catch i8* null
  %0 = load i64, i64* @RestoreCSVBitMask
  %1 = and i64 %0, 0x1
  %2 = icmp eq i64 %1, 0            ; C1 = (RestoreCSVBitMask & RDI)
  %3 = load i64, i64* %rdi_local
  %4 = icmp ne i64 %3, 0xDEADBEEF   ; C2 = (rdi_local != 0xDEADBEEF)
  %5 = and i1 %2, %4                ; Res = !C1 && C2
  %6 = load i64, i64* @rdi          ; if (Res) rdi = rdi_local
  %7 = select i1 %5, i64 %3, i64 %6
  store i64 %7, i64* @rdi           ; else rdi = rdi
  %8 = or i64 %0, 0x1               ; if (Res)
  %9 = select i1 %5, i64 %0, i64 %8;   RestoreCSVBitMask |= RDI
  store i64 %8, i64* @RestoreCSVBitMask
  ; Repeat for all the CSVs. At the end, propagate the exception.
  resume { i8*, i32 } %lpad
```

Figure 4.4: The csv_restore basic block, installed on each isolated function and taken whenever the function throws an exception or it is being propagated.

Coupled with this mechanism, EnforceABI is complete and the whole rev.ng pipeline is shown in Figure 4.7.

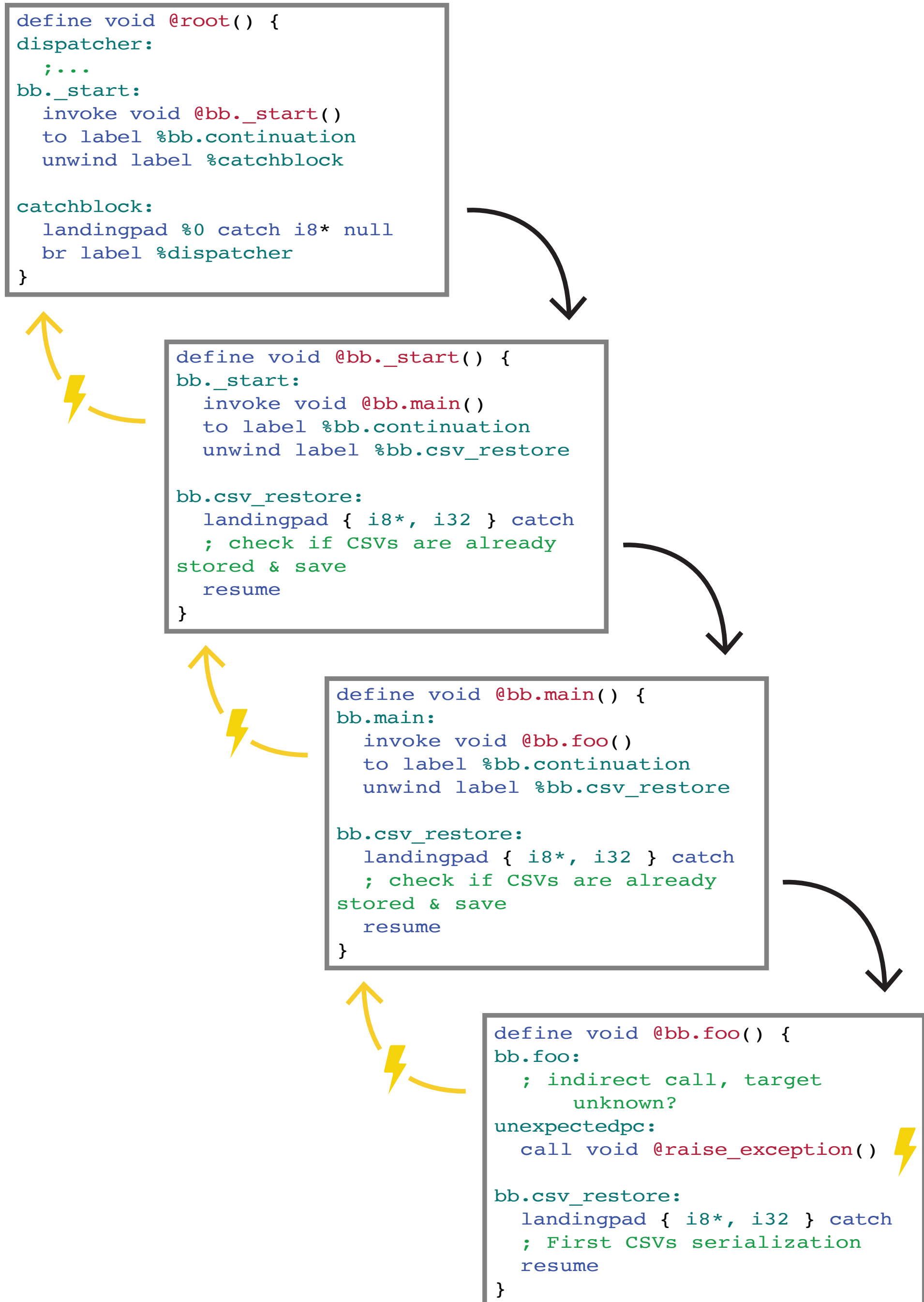

define void @root() {
dispatcher:
;...
bb._start:
invoke void @bb._start()
to label %bb.continuation
unwind label %catchblock
catchblock:
landingpad %0 catch i8* null
br label %dispatcher
}
define void @bb._start() {
bb._start:
invoke void @bb.main()
to label %bb.continuation
unwind label %bb.csv_restore
bb.csv_restore:
landingpad { i8*, i32 } catch
; check if CSVs are already stored & save
resume
}
define void @bb.main() {
bb.main:
invoke void @bb.foo()
to label %bb.continuation
unwind label %bb.csv_restore
bb.csv_restore:
landingpad { i8*, i32 } catch
; check if CSVs are already stored & save
resume
}
define void @bb.foo() {
bb.foo:
; indirect call, target unknown?
unexpectedpc:
call void @raise_exception()
bb.csv_restore:
landingpad { i8*, i32 } catch
; First CSVs serialization
resume
}

Figure 4.5: Example of execution flow of a program translated and optimized with EnforceABI. Suppose that the flow of the original program starts with `_start` → `main` → `foo`. This latter function has an indirect call to another method, whose target was not statically determined, and matches no one of the basic block targets provided by `function_dispatcher`, previously unrolled. At this point, the only possible basic block to be taken is `unexpectedpc`, which throws an exception. During the stack unwinding, the first handler to be called is `foo`'s `bb.csv_restore`, where the CSVs clobbered by `foo` are serialized. Then, the exception is passed to the caller, `main`, which serializes the leftover CSVs. The propagation goes on until `root` is reached, whose `catch`-block ends the exception and jumps to the dispatcher; there, the target of the indirect call will be finally resolved.

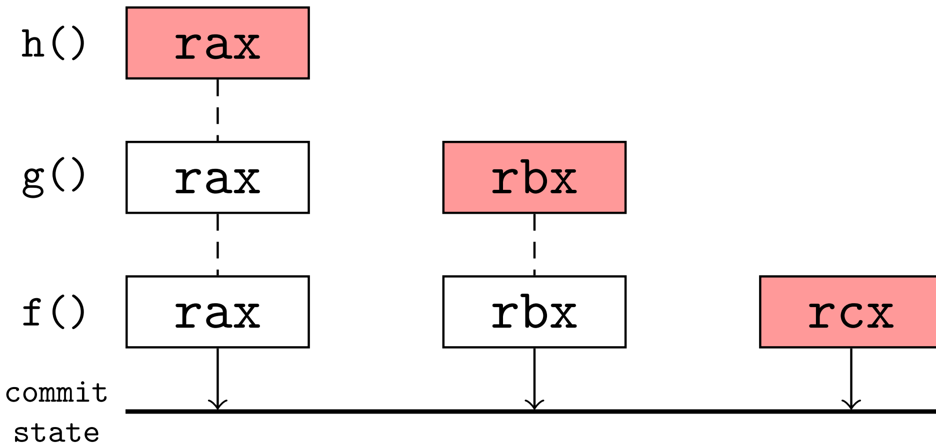


Figure 4.6: Commit state during stack unwinding: in horizontal, the registers overwritten by the corresponding function. The local CSVs clobbered by the innermost frame (`h`'s one) are serialized into the global CSVs first (marked in red) while its exception handler is being executed. The unwinding process continues until the last handler (`f`'s one) before `root` executes, which serializes the leftover CSVs (`rcx`).

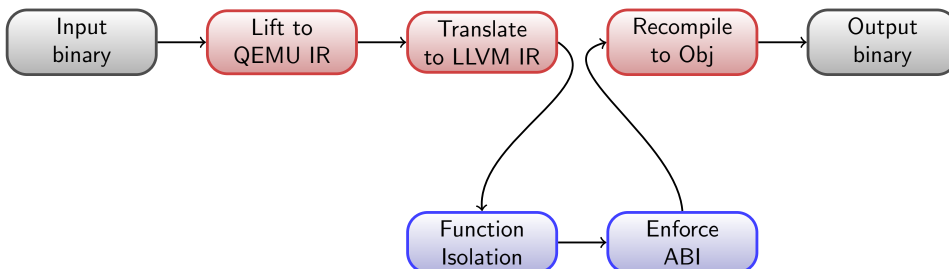


Figure 4.7: The final `rev.ng` pipeline, now both with Function Isolation and EnforceABI passes that run on the LLVM IR module.

Chapter 5

Experimental Evaluation

This section is dedicated to the experimental results we obtained to evaluate the effectiveness of `rev.ng-fuzz`. Typically, a good approach to measure ground truth is to assess fuzzers against known bugs or through a synthetic flawed suite (e.g., LAVA [16]). We decided to carry out a case study on a real-world vulnerability, by applying `rev.ng-fuzz` to fuzz an old version of `less`, the UNIX terminal pager, whose regular expression library linked against is known to suffer from a stack-based buffer overflow vulnerability. The bug was assigned CVE-2015-3217 [11]. Also, it is particularly interesting to investigate the performance when comparing our framework to another state-of-the-art in fuzzing binaries, i.e., AFL in QEMU-*mode*. The fact that AFL is a highly effective and radically different approach makes it the best candidate to compare with. We show that, not only is `rev.ng-fuzz` able to find successfully the bug, but also significantly outperforms AFL-QEMU.

On the ABI enforcement side, the second use case is concerned with testing the execution of a translated binary optimized with our improvements on `EnforceABI`. We decided to lift to LLVM IR a simple program, a calculator written in C, recompile and run it, in order to benchmark the results of multiple program executions. The first time the LLVM module is optimized only with the Function Isolation transformation pass; afterwards, it goes through EnforceABI as well (see Section 4.2).

5.1 Reproducing CVE-2015-3217

To assess the efficacy of `rev.ng-fuzz`, we built `less` by compiling its source code to a static executable for GNU/Linux on x86-64. We specifically took a flawed version of the Perl Compatible Regular Expression (PCRE) library [23], which is statically linked against `less`. We can put ourselves in the same position of an analyst looking at the program without any a priori knowledge (*blackbox setting*) by stripping the symbols from the executable. Note that we could not achieve the same scenario if we were to fuzz directly the PCRE library, since it is not possible to strip the dynamic symbol table from a shared object without making it unusable (given that the functions would be no longer exported).

After a reverse engineering phase, the relevant `libpcre` targets identified to be fuzzed turn out to be the following ones.

`pcre_compile`: it takes a string, compiles it to a regular expression object, and returns it.

`pcre_exec`: it runs the previously compiled regular expression on a string.

`free`: it frees the compiled regular expression object.

The next step consisted in determining the entrypoint to start fuzzing: in our case, it was the `less`' `main` function, so that the initialization routines can execute properly. Once `less` has been lifted with `rev.ng`, we are left with putting in harness the functions to stimulate, i.e., the above three functions. Listing 5.1 shows how the implementation of the `LLVMFuzzerTestOneInput` looks like. Recall from Section 3.2.3 that the `srtf_*` functions are the generated wrappers around the isolated functions, provided by `rev.ng-fuzz` and directly callable from the C header, to be included from the runtime support library.

After linking the runtime support library and the LLVM IR module together, and recompiling it all, we ran the translated program. The execution semantics is the same as the original program till the entrypoint is met, and then the harness function takes control (see Section 3.3). We obtained a crash few seconds later, managing to reproduce the vulnerability. Inspection of the binary at the crashed location confirmed the presence of the stack-overflow.

Note that, prior to extending the `rev.ng` capabilities with automated fuzzing, calling functions required to manually prepare arguments in the appropriate registers. An example is illustrated in Listing 5.2, which shows how the function call to `pcre_exec` used to be done [18].

```
// Run pcre_exec to run the compiled regex
pc  = Address_pcre_exec;
rdi = Result;
rsi = 0;
rdx = (uint64_t)&str_to_match;
rcx = (uint64_t)strlen(str_to_match);
r8  = 0;
r9  = 0;
*stack = OVECTOR_SIZE; stack--;
*stack = (uint64_t)&ovector; stack--;
root(stack);
```

Listing 5.2: The manual preparation of the CSVs to call functions to fuzz before the introduction of `rev.ng-fuzz`. Here the call to `pcre_exec` is shown. As it can be seen, the program counter is assigned the on-disk virtual address of the call (manually identified). All the arguments need to be passed by hand on the specific registers following the target calling convention (i.e., x86-64); with the remaining ones pushed onto the emulated stack in reverse order (last one, first). Clearly, the stack pointer needs to be manually decremented as well. Finally `root` is called.

```c
#include "generated_prototypes.h"

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // Regexp to be mutated.
  char regexp[1000] = "(?:(?(1)1|";

  // While not strictly necessary, this simple mutation of the
     regexp allows to slightly reduce the bug discovery time.
  size_t len = strlen(regexp);
  for (size_t i = 0; i < size; ++i) {
    char c = '1';
    switch ((data[i] >> 3) & 7) {
      case 1:
        c = '('; break;
      case 2:
        c = ')'; break;
      case 3:
        c = '|'; break;
      case 4:
        c = ':'; break;
      case 5:
        c = '+'; break;

      // Other cases handled here.
        default: ;
    }
    regexp[len++] = c;
  }

  regexp[len++] = '\0';
  const char str_to_match[] = "Test string";

  // Wrappers being called here.
  uint64_t compiled_re = srtf_pcre_compile((uint64_t)&regexp, 0,
     (uint64_t)&errorstring, &erroroffset, NULL);

  #define OVECTOR_SIZE 3
  int ovector[OVECTOR_SIZE];
  srtf_pcre_exec(compiled_re, NULL, (uint64_t)&str_to_match,
     sizeof(str_to_match), 0, 0, &ovector, OVECTOR_SIZE);

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

Listing 5.1: The implementation of the harness function when exercizing the identified targets within `less`, i.e., `pcre_compile`, `pcre_exec` and `free`. All of these functions are called through their respective wrappers by including the appropriate header generated by `rev.ng-fuzz`.

Performance. To measure the fuzzing rates of our framework, we executed `rev.ng-fuzz` and AFL separately and compared the performance in terms of executions per second. Note that to simulate the same scenario with AFL, we slightly changed the setting: instead of building `less`, we built a small program, linked against the PCRE library, that would exercize the three above functions and fed the input to AFL. This change stems from the fact that AFL would need to spawn `less` at each iteration of fuzzing, which would end up being detrimental for performance, besides being too unfair. Note that this performance exacerbation could theoretically be alleviated by employing AFL in *persistent mode*; however, it is not available when instrumenting the binary under QEMU.

By contrast, in `rev.ng-fuzz` calling `free` in the `LLVMFuzzingTestOneInput` routine to release the memory allocated by `pcre_compile` for the PCRE object is mandatory otherwise in order to avoid experiencing soon a memory shortage. This is not an issue for AFL: in fact, due to forking every time, the virtual memory of the child process, at its termination, is automatically unmapped and reclaimed by the kernel.

The results are reported in Table 5.1. As it can be seen, `rev.ng-fuzz` outperforms AFL in QEMU-*mode* with speedups ranging from 20× to 40×. As time goes by, `rev.ng-fuzz`'s performance slightly deteriorate due to the exploration of new paths progressively leading towards more complex inputs, whereas AFL's exploration remains mostly constant.

Nonetheless, the overall number of executions of `rev.ng-fuzz` prevail over the ones of AFL, allowing a larger input space to be explored, thus confirming that, not only has static binary analysis a great impact on binary fuzzing, but also ease and improve the analyst's workflow considerably.

	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-fuzz`	150 617	79 701	78 306	271 217 728

Table 5.1: Comparison of the performance between AFL used in QEMU-*mode* and `rev.ng-fuzz` when fuzzing on user-defined entry points. 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.

5.2 Running a program with `EnforceABI`

We evaluate the enhancements over EnforceABI by running a simple program, a calculator, that has been translated and optimized with EnforceABI. We measure 5 different executions considering the wall-clock of the process with the UNIX utility `time(1)`. The calculator is wrapped in a loop that runs 10000000 times, it increments the result value every time, which is finally printed. The first 5 runs are computed with only Function Isolation pass, whilst, in the other five, the LLVM IR has been optimized with EnforceABI as well. Note that, at this time, only the CSVs corresponding to the registers `rsp`, `rbp`, and `pc` are not promoted to local variables. Table 5.2 reports the results of our experiments.

	I	II	III	IV	V	Average
Function Isolation	13.604	13.604	13.609	13.607	13.610	13.606
EnforceABI	12.925	12.935	12.928	12.927	12.934	12.929

Table 5.2: Comparison of performance of the execution of `calc` before and after running `EnforceABI`. Time is measured in seconds. In both cases, the binary has been optimized with `O2` level. The binary optimized with EnforceABI, on average, runs 5% faster.

As it can be seen, the binary translated with the EnforceABI pass is prone to more optimizations, achieving a speedup of 1.05×. We do expect to obtain a bigger speedup in larger test suites, with binaries where the register pressure is higher. Also note that, while the CSVs serialization introduced to make `EnforceABI` runnable is an expensive procedure, in practice it is executed only when an exception is thrown, i.e., in rare cases of control-flow recovery inaccuracy.

Conclusions

In this work, we introduced a novel framework that allows to perform sanitized coverage-guided fuzzing of individual functions of closed-source executable binaries. We have seen that the main advantage of performing fuzzing with `rev.ng` is that one can easily write the harness function (`LLVMFuzzerTestOneInput`) in C/C++ by calling the recovered functions of the binary with low performance impact – as though we were in a whitebox scenario.

Then, in order to improve the runtime, we focused on enhancing an analysis that enforces the ABI by promoting the set of variables representing the CPU state (CSVs) from global variables to function arguments and local variables. In general, this allows to catch more optimization opportunities and therefore, leading to fewer memory accesses to perform. This has required the design and implementation of a safeguard mechanism that allows execution of the translated binary while preserving its functional correctness in case of exceptions due to control-flow recovery inaccuracies or limitations intrinsic to static analysis (e.g., undetermined jump target in indirect calls). We deem that exploiting these execution performance enhancements may be particularly beneficial to improve the fuzzing rates.

We are currently revisiting an analysis (*Stack Analysis*) that aims at identifying the function arguments and return values of the recovered functions in a ABI-agnostic way, in order to improve the quality of the function prototypes generated for fuzzing. Note also that, as of now, variadic functions are supported by manually passing them the number of arguments to be pushed onto the stack. This constraint can be relaxed in the future.

In our upcoming work, we plan an ABI-specific layer to further refine the results of previous analyses as well as to add support for further architectures. We also intend to perform further benchmarking and comparisons against other recent state-of-the-art binary fuzzers (e.g., `afl++` in QEMU-*mode* [20]). We may extend integration to LLVM-based symbolic execution engines (e.g., KLEE [8]) as well. On the ABI enforcement side, we plan to measure the performance improvements of programs on the SPECint 2006 benchmarks suite [24].

Finally, a blog post with a demo of this work was published [2], piquing the interest of a major player in the information security industry.

Bibliography

[1] C++ ABI for Itanium: Exception Handling. https://refspecs.linuxbase.org/abi-eh-1.21.html.

[2] Fuzzing binaries with LLVM's libFuzzer and rev.ng. https://rev.ng/blog/fuzzing/post.html.

[3] SanitizerCoverage: Clang documentation. https://clang.llvm.org/docs/SanitizerCoverage.html.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[5] Andrea Biondo. Improving AFL's QEMU mode performance, 2018. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[6] Apple. Advanced Debugging and the Address Sanitizer. https://devstreaming-cdn.apple.com/videos/wwdc/2015/413eflf3lrh1tyo/413/413_advanced_debugging_and_the_address_sanitizer.pdf.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[9] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[10] Cristina Cifuentes and Vishv Malhotra. Binary translation: Static, dynamic, retargetable? In *Software Maintenance 1996, Proc., Int. Conf. on*, pages 340–349. IEEE, 1996.

[11] National Vulnerability Database. NVD - Detail - CVE-2015-3217. https://nvd.nist.gov/vuln/detail/CVE-2015-3217.

[12] Christophe de Dinechin. C++ Exception Handling for IA64. In *First Workshop on Industrial Experiences with Systems Software (WIESS 2000)*, San Diego, CA, October 2000. USENIX Association.

[13] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 15–27, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 131–141, New York, NY, USA, 2017. ACM.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[16] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[17] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the 2016 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '16, Piscataway, NJ, USA, Oct 2016. IEEE Press.

[18] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery. In *2018 International Carnahan Conference on Security Technology, ICCST 2018, Montreal, QC, Canada, October 22-25, 2018*, pages 1–5. IEEE, 2018.

[19] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[21] Google. Honggfuzz: Security oriented software fuzzer. `https://github.com/google/honggfuzz`.

[22] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. Performance, Correctness, Exceptions: Pick Three. A Failproof Function Isolation Method for Improving Performance of Translated Binaries. In *Workshop on Binary Analysis Research*, BAR '19, Feb 2019.

[23] Philip Hazel. PCRE – Perl Compatible Regular Expressions. https://www.pcre.org.

[24] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[25] Paul Hernault. Fuzzing binaries using Dynamic Instrumentation. https://project.inria.fr/FranceJapanICST/files/2019/04/19-Kyoto-Fuzzing_Binaries_using_Dynamic_Instrumentation.pdf, 2019.

[26] R. Nigel Horspool and Nenad Marovac. An Approach to the Problem of Detranslation of Computer Programs. *Comput. J.*, 23(3):223–229, 1980.

[27] Reid Kleckner and David Majnemer. Exception handling in LLVM, from Itanium to MSVC. https://llvm.org/devmtg/2015-10/slides/KlecknerMajnemer-ExceptionHandling.pdf.

[28] Konstantin Serebryany. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, 2016.

[29] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004.

[30] Peter Maydell and QEMU Team. QEMU.org. https://www.qemu.org/.

[31] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[32] Alexandru-Vlad Niculae. Enabling coverage-guided binary fuzzing on macOS. https://github.com/AlexNiculae/PoCxZer0Con-TinyInst-Slides.

[33] James Oakley and Sergey Bratus. Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, page 11, USA, 2011. USENIX Association.

[34] Quarkslab. QBDI: QuarkslaB Dynamic binary Instrumentation. https://qbdi.quarkslab.com.

[35] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings 2017 Network and Distributed System Security Symposium*, 2017.

[36] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer, 2018. https://github.com/MozillaSecurity/dharma.

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[38] Konstantin Serebryany, Vitaly Buka, and Matt Morehouse. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator. 2017.

[39] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 1st edition, 2008.

[40] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[41] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 627–642, USA, 2015. USENIX Association.

[42] Michał Zalewski. American Fuzzy Lop - Whitepaper. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016.

[43] Google Project Zero. Jackalope: Binary, coverage-guided fuzzer for Windows and macOS. `https://github.com/googleprojectzero/Jackalope`.

[44] Google Project Zero. TrapFuzz. `https://github.com/googleprojectzero/p0tools/tree/master/TrapFuzz`.