Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Doctoral Programme In Computer Science and Engineering

Compiler Techniques for Binary Analysis and Hardening

Doctoral Dissertation of:
Alessandro Di Federico

Supervisor:
Prof. Giovanni Agosta

Tutor:
Prof. Andrea Bonarini

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

XXX Cycle

Contents

List of Figures

List of Algorithms

List of Listings

List of Tables

Abstract

Despite the growing popularity of interpreted or byte-compiled languages, C/C++ and other languages targeting native code are still dominantly used for system programming. Programs compiled to native code present a set of challenges compared to alternatives. In particular, in this work we focus on how they can be efficiently analyzed, how existing security measures (known as *binary hardening techniques*) perform, and how new ones can be introduced to secure features that have received little attention.

We propose `rev.ng` a binary analysis framework based on QEMU, a popular dynamic binary translator and emulator, and LLVM, a mature and flexible compiler framework. `rev.ng` can easily handle a large number of architectures and features a set of analyses to recover basic blocks locations, function boundaries and prototypes in an architecture- and ABI-independent way. `rev.ng` can be used for instrumentation, debugging, decompilation, retrofitting of security features and many more purposes. Our prototype encompasses about 17 kSLOC of C++ code and has been publicly released under a Free Software license.

The core component of `rev.ng` is `revamb`: a static binary translator which can accurately identify all the basic blocks, and, in particular, the targets of indirect jumps for `switch` statements. Along this work, we will make heavy use of analysis techniques popular in the compiler literature, such as Monotone Frameworks, to recover an accurate control-flow graph, identify function boundaries and the number and location of function arguments and return values.

We will also discuss how `rev.ng` can handle native dynamic libraries, how it can be easily employed for instrumentation purposes, how it can be extended to handle even more architectures and how its performance compares to tools with analogous purposes such as QEMU, Valgrind, Pin and angr.

We also study two often overlooked features of C/C++ programs: variadic functions and the RELRO link-time protection mechanism. We propose `HexVASAN`, a sanitizer for variadic functions to ensure that the number and type of arguments used by the variadic function match those passed by the caller, and `leakless`, an exploitation technique to bypass the RELRO protection in its several forms.

Introduction

Despite the growing popularity of interpreted languages, such as Python and Ruby, and languages that compile to some form of byte-code, such as C# or Java, a large part of the software is still being written in languages directly targeting *native code*, such as C and C++. Such languages are popular for performance critical software, system level software, bare metal software, kernels and embedded systems in general.

Compared to languages targeting interpreters or byte code, they generally provide better performance but carry with them a series of challenges. In this work we will focus on two challenging aspects in particular: reverse engineering and safety of native code.

On the reverse engineering side, which in the native world takes the name of *binary analysis*, we face many issues: native code is not intended to be read by a human or even a compiler. It's a final product, ready to be run, which is not supposed to be further manipulated (e.g., for optimization purposes) or read. Native code is composed of low level instructions which provide basically no abstraction whatsoever. A loop might be completely unrolled, common strings of assembly instructions might be factored out in a function, arithmetic or logical operations might be turned into different but semantically equivalent instructions and so on.

On the safety side, languages targeting native code often do not provide full memory safety, which makes it harder to write secure software. Therefore, along the years, a wide range of countermeasures have emerged. Many of such protection mechanisms can be automatically applied during the compilation process without further action on the developer side. Some of the most popular ones include ASLR, CFI, stack canaries, W^X and many others. These techniques, which involve the compiler, the linker, the dynamic loader, the kernel and other components of the operating system, are collectively know as *binary hardening techniques*.

Binary Analysis

Digital devices have become a key part of our everyday life, carrying a large amount of benefits, and risks. In fact, for each software platform gaining enough popularity, a set of malicious software (malware) to abuse them is developed.

To be able to effectively oppose them, understanding their behavior and how they're built is of critical importance. For this reason, a series of tools for analyzing software for which source code is not available is fundamental. Such tools are often known as *reverse engineering* or *static binary analysis tools*. In certain situations, the reverse engineering effort can be aided by not only analyzing the program, but also running it with certain changes to better understand its behavior. Such an activity is known as *dynamic binary analysis*.

Information learned through static analysis is generally applicable but runs into precision issues while dynamic analysis has better precision, as an actual instance is being evaluated, but is limited to existing test cases, i.e., the code must be *executed* in a run-time setting to be analyzed.

Reverse engineering tools can also be employed to analyze ordinary software, typically provided by third parties which do not release the source code. The analysis of non-malicious software enables the assessment of the security of components outside the control of their users. For instance, companies or public entities dealing with sensitive data might want to analyze software provided by third parties to investigate the presence of bugs or, even worse, backdoors. To these ends, the most useful reverse engineering tool is a *decompiler*, a tool able to recover a representation of the program's behavior that is close to the original source code, and therefore easier to understand with respect to low level machine code which is produced by disassemblers.

While several reverse engineering tools are available, they have been lacking in innovation, usability and the diversity of the hardware platforms they are able to handle. In fact, existing tools either develop a set of ad-hoc heuristics to handle each new architecture (IDA Pro [72]) or, if the tool employs an intermediate representation for its analyses, it has to create a new front-end to handle each new architecture. Examples of the latter case are BAP/ByteWeight [20, 11] (x86 and ARM only), MC-Semantics [152] (x86 only) and LLBT [139] (ARM only).

The diversity aspect is of particular relevance in a world being progressively dominated by IoT devices. In fact, the IoT ecosystem has historically paid little to no attention to security aspects and often employs unusual architectures, which makes the software running on them harder to analyze due to the lack of tools for those architectures. Therefore, the analyst is left with severely insufficient analysis capabilities. This is particularly important in the context of eHealth devices which are critical for the life of an always growing number of people and often employ little known, ultra-low power CPUs (e.g., RISC-V [175]).

In this work we propose `rev.ng`, a unified system for binary analysis which employs a set of principled techniques rather than architecture-specific heuristics for its analyses and that, unlike existing works, defers the burden of providing a reliable front-end for a wide range of architectures to existing tools. In our prototype we rely on QEMU to provide such a front-end. QEMU is a dynamic binary translator that lifts binary code into a custom intermediate representation (IR) for 17 different architectures, including x86, x86-64, MIPS, ARM, and AArch64. As a tool aiming at full system emulation, QEMU supports even the most sophisticated ISA extensions. For instance, it already supports the recently introduced Intel MPX ISA extensions. Moreover, the large community and industry interest around QEMU virtually guarantees that new architectures and ISA extensions are supported promptly (e.g., RISC-V [129]).

The main benefit of this approach, besides the vastly reduced effort in handling a large number of ISAs, relies on the fact that all the supported architectures are first-class citizens. Even more importantly, all the analyses built on top of the QEMU IR have to be designed in an ISA-agnostic way. In practice, when handling ISA-specific peculiarities, such as MIPS delay slots or ARM predicate instructions, the approaches are equally effective, respectively, on OpenRISC delay slots and x86-64 conditional move instructions. Many of the analyses that we will present apply techniques well known in the compiler field, such as data-flow analysis.

In our implementation, `rev.ng`, we translate the IR provided by QEMU into LLVM IR, an environment that facilitates further analyses. On top of this, `rev.ng` does not only allow the analyst to understand what a piece of software does, but also to alter its behavior in an easy way and introducing a low overhead compared to alternative tools. Therefore rev.ng also provides a unified framework for both *static* and *dynamic analysis*.

In the context of binary analysis, when dealing with C/C++ programs stripped of debugging information, there are a series of challenging tasks. In Part I, after providing the necessary background (Chapter 1), in Chapter 2, we will introduce `rev.ng`, our static binary analysis framework. In Chapter 3 we will focus on the identification of all the basic blocks in the code. In Chapter 4 our approach for the recovery of an accurate *control-flow graph* for the program and the detection of the boundaries of the original functions will be presented. Chapter 5 will introduce the `rev.ng` function prototype detection algorithm, i.e., the recovery of function arguments and return values. This information is vital for any reverse engineering efforts, and to build additional tools such as a decompiler. Part I will conclude with a review of related works (Section 6.2).

Binary Hardening

In Part II, we analyze two often overlooked aspects of the security of a binary program: the RELRO link-time hardening and variadic functions.

RELRO is a protection mechanism applied by the linker which comes in two flavors: partial and full. Its purpose is protecting data structures used by the dynamic loader to resolve functions. In fact, such data structures can be abused by an attacker able to run code (e.g., through a ROP chain) in the target process to call any library function. This is particularly interesting for the attacker since it reduces the complexity of the attack and allows him to call library functions such as `system` or functions more involved in the behavior of the specific program. In Chapter 8 we will describe and exploit several key design flaws in RELRO and suggest countermeasures.

The other key aspect of binary programs we focused our attention on are variadic functions. They are a popular feature of C/C++ programs which offers great flexibility. The most popular variadic function is the `printf` function. Such a flexibility comes at the cost of a non-negligible loss in terms of type safety. This is true at the point that a large part of a whole class of vulnerabilities, format string attacks, is due to this unsafety.

Variadic functions accept an arbitrary number of arguments of arbitrary types. If an attacker is able to provide arguments in an unexpected number or type, it can corrupt the integrity of the execution environment, with possibly serious security consequences. In Chapter 7, we design and test the effectiveness of a compiler-based countermeasure for such attacks. The countermeasure is implemented as an LLVM sanitizer which instruments the code of variadic functions and variadic function calls, so that, at run-time, before accessing each argument, a check is performed to ensure that such an argument has actually been provided, and that the expected type corresponds too.

Apart from these two aspects, in terms of binary hardening, we also introduce a static binary translator based on the binary analysis framework presented in the previous section. Binary translation is a technique that, given an executable program (or a portion of it) compiled for a certain architecture (e.g., ARM), aims to translate it into a different one (e.g., x86-64). Binary translation can be performed statically or dynamically. The dynamic version is essentially a type of emulation technique where the emulation, usually a slow process, is sped-up by translating instructions into the host machine code at run-time. The static version is a much more complex endeavor, and it is aimed at building a full executable program which can then be executed autonomously. The motivation for developing a binary translator is provided by a wide range of applications. Historically, legacy code performance portability was the key motivation. Indeed, binary translation techniques have been employed to provide binary compatibility for new platforms, such as the Transmeta Crusoe [45] which achieves better performances than an emulator. In our cases, one of the main use cases consists in retrofitting binary hardening countermeasures, absent in the original binary, such as CFI [173, 156, 118, 170].

Static binary translators are particularly interesting because they do not impose a run-time overhead as significant as other approaches, and they can be used to generate new binaries that are completely independent of the translation system. However, they pose a number of additional challenges which must be solved in order to produce a standalone binary program.

In particular, discovering and identifying code in a binary object is a non-trivial task. Ideally, one would start from the entry point of the program, follow all jump and call to subroutine instructions to identify the starting points of all reachable code segments. However, this is made more complex by the usage of function pointers, virtual functions and, most importantly, `switch` statements. Chapter 2 and Chapter 3 describe our approach to these issues.

Part I

rev.ng: a unified binary analysis framework

Chapter 1

Background

In this chapter we will present the key concepts and tools required to have a proper understanding of the rest of Part I. Specifically, we will first provide an overview of the compilation process, in general, and, specifically, for the ELF image format. Understanding the compilation process is fundamental, since our final aim is to *reverse* it. Then we will dig into the LLVM compiler framework (and its IR in particular) and the QEMU internals since they constitute the mainstays of `rev.ng`. The chapter ends with a section introducing Monotone Frameworks, a key theortical tool for building many of the analyses we will present to recover abstractions from binary code.

1.1 The Compilation Process

A large part of this work is devoted to the analysis of binaries. It's therefore of great importance to understand how such binaries are produced. In this section we will offer an overview of how a modern compiler works, from the source code down to the final executable binary.

As shown in Figure 1.1, the compilation toolchain for languages such as C and C++, our main focus, is typically divided in four major components: the preprocessor, the compiler, the assembler and the linker.

The preprocessor. The preprocessor is the component responsible for, given an input file, removing comments, enabling or disabling certain portions of code (through the `#ifdef` directives) and, most importantly, expanding its body with all the included files (`#include` directives) and macros (`#define` directives). The final output is pure C code suitable to be fed to the compiler, which is not supposed to handle the above mentioned directives.

The compiler. The compiler is the core component of the compilation toolchain. It takes as input a preprocessed source file, also known as *translation unit*, and produces an assembly file, containing data and functions almost in their final form, i.e., instructions for the target CPU. Internally, modern compilers are divided in three further components: the *front-end*, the *mid-end* and the *back-end*. The *front-end* is programming language-specific: it parses the input file, builds an AST and produces an intermediate representation (IR) which is independent from the input language and mostly independent of the targeted machine. The *mid-end* performs a set of "high-level" optimizations that are independent from the target architectures, such as loop unrolling, inlining and outlining of functions and many more. Finally, the *back-end* transforms the IR into a target-specific representation (i.e., where the functions contain instructions that are specific for a certain architecture) and performs another set of low-level optimizations that make sense or are beneficial only for the target CPU.

The assembler. The assembler takes as input a file with a textual representation of the functions, instructions and global data and encodes them into their binary representation, often referred to as *machine code*, that the CPU can understand. The output file is known as the *object file*. Note that the code is not in its final form. In fact, references to external variables or functions, and, in certain conditions, even references to the parts of functions defined in the current translation unit, are left pending, since only the linker can know their final value.

The linker. The linker is the component responsible to collect multiple object files, decide how their data and code is laid out (e.g., the code of the second object file is appended to the one of the first one), and *fix* references to code and data, left pending by the assembler, with their final value. In fact, the final position of each function or piece of data is known exclusively by the linker, which is the only component of the toolchain which has global visibility on all of the object files composing the program being compiled. The linker is allowed to leave some references pending, in case the corresponding function or data are available in a dynamic library. Another component, the dynamic loader, will finish the linking process upon program startup.

1.2 ELF

ELF (Executable and Linkable Format) is an image format for object files, executable programs and dynamic libraries. It's the standard image format on GNU/Linux and many other unix-like operating systems (with the notable exception of macOS) and for embedded systems. In this section we will provide an in-depth description of it, since it's the main image format we consider in this work. Note however that other binary formats (such as Mach-O) have analogous features and extending a tool to support them is primarly engineering work.

Each ELF file starts with the ELF header. The ELF header contains the following information: a magic byte sequence used to easily identify the file as an ELF image, its the endianess, the target architecture, whether it's a 32-bit or 64-bit, the target operating system and ABI, the type of ELF image and a couple more fields.

The ELF format is defined (in terms of the data structure it contains and their layout) independently from the target architecture, except for two characteristics: the endianess and whether the target architecture is 32- or 64-bit. Therefore, depending on the size and the endianess used to encode integers, there are four different type of ELF formats: ELF32BE, ELF32LE, ELF64BE, ELF64LE. Except for this, and other minor differences, they are analogous.

There are five different types of image formats that can be encoded in ELF:

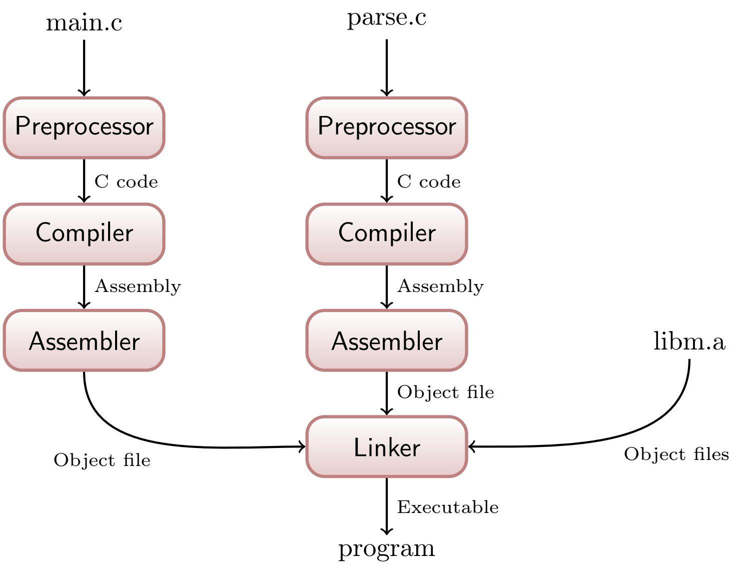


Figure 1.1: Overview of the compilation process.

Object files (`ET_REL`) *Relocatable* object files, as emitted by the compiler/assembler. They typically have a `.o` extension.

Executable programs (`ET_EXEC`) Executable programs, linked and ready to be run. They usually have no extension.

Shared objects (`ET_DYN`) A dynamic library, linked and ready to be loaded by an executable program. They typically have a `.so` extension.

Core files (`ET_CORE`) Images of a process that was running, written on disk for debugging purposes.

Unknown (`ET_NONE`) An ELF file of an unspecified type.

In the following, we'll provide an overview of the key features of object files, executable programs and we'll conclude with an overview of how dynamic libraries and the dynamic loading process works.

1.2.1 Object Files

As previously mentioned, the assembler produces object files. ELF object files are primarily composed by three parts:

Sections. A section is a portion of the ELF object file that can contain either code or data. They are described by a *section header* that contains an offset within the file where the section begins, its size, a type and a series of flags. The most interesting sections to our ends are:

`.text` The core section containing the code of the object file. This section contains relocatable (i.e., with unresolved pending references) machine code.

`.data` The writeable global data of the translation unit.

`.rodata` The read-only global data of the translation unit.

`.bss` The writeable global data of the translation unit. This section differs from `.data` in the fact that data here is "uninitialized", which means, automatically initialized to zero. Since the content of this section is implicitly all zero, it's not associated to any data in the object file. Therefore, its starting offset in the file has no real meaning and the size is just an indication of how much space will be needed (and zero-initialized) at run-time.

Symbol table. Another key section of object files is `.symtab`, which contains all the *symbols* of the translation unit. A symbol is a label for a portion of data or code, with an optional size. In practice, a symbol typically represents either a global variable or a function. If the symbol is defined in the current translation unit, then the symbol is also associated to an offset within the object file where it starts. Otherwise, if the symbol is used but not defined in the current translation unit, it is marked as undefined.

Relocations. The last core component are relocations. A relocation is basically a directive for the linker instructing it to write the value (i.e., address) of a certain symbol in a certain position within a section. Each one of the sections listed above can be associated with a section containing a sequence of relocations that apply to it. Such sections have the same name as the target section with a `.rel` prefix (e.g., relocations for `.text` will be in the `.rel.text` section).

Depending on the architecture, a relocation could also include a constant addend to add to the value of the symbol. In such cases, the relocations will have a `.rela` prefix.

It is important to understand that a relocation doesn't always write the address of the associated symbol in the same way. For example, in certain situations a relocation might instruct the linker to write the full 32-bit address of the symbol in a certain position, while in other situations it might require the value to be written as an offset from the destination address. This is particularly useful to encode the offset of a PC-relative jump instructions.

Listing 1.1 shows the relevant parts of an object file.

1.2.2 Executable Programs

As illustrated in Section 1.1, the linker is responsible to produce the final executable, performing the following steps:

Lay out the program. The linker parses all the input object files, and decides how the various sections have to be laid out in the final program. In practice, it groups the input sections into *segments* based by their attributes, such as being writeable, readable or executable. For instance, this means that `.data` and `.bss` will be grouped together since they are both writeable but not executable. Then, it concatenates all the input sections with the same name from the various input object files. Such an operation is fundamental to be able to assign a final address to each symbol.

```
$ gcc main.c -c -o main.o
$ readelf -S main.o
There are 7 section headers, starting at offset 0x130:

Section Headers:
  [Nr] Name      Type     Addr     Off    Size   ES Flg Lk Inf Al
...
  [ 2] .text     PROGBITS 00000000 000040 00002f 00  AX  0   0 16
  [ 3] .rel.text REL      00000000 0000e0 000008 08      6   2  4
...
  [ 6] .symtab   SYMTAB   00000000 0000a0 000040 10      1   2  4

$ readelf -s main.o

Symbol table '.symtab' contains 4 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
...
     2: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND function
     3: 00000000    47 FUNC    GLOBAL DEFAULT    2 main

$ objdump -d main.o
...
Disassembly of section .text:

00000000 <main>:
...
  1e:   89 04 24                mov    DWORD PTR [esp],eax
  21:   e8 fc ff ff ff          call   22 <main+0x22>
  26:   8d 44 06 03             lea    eax,[esi+eax*1+0x3]
...

$ readelf -r main.o

Relocation section '.rel.text' at offset 0xe0 contains 1 entries:
 Offset     Info    Type            Sym. Value  Symbol's Name
00000022  00000202 R_386_PC32        00000000   function
```

Listing 1.1: Key parts of an object file. main.o is the C example in Listing 1.3 compiled for x86 to an object file. Note that in the main function we had a call to function, which was not defined in the given source file. readelf -S lists the sections contained in the object file. We can see the symbol table (.symtab), the code section (.text) and its relocations (.rel.text). readelf -s lists the symbols defined in the symbol table. Note that the main symbol is defined at offset 0 from section 2 (.text), while the function symbol is marked as undefined (UND). objdump -d shows the relocatable assembly code of the main function. Note that the address of the target of the call function is not available. readelf -r lists the relocations. The only relocation is relative to the (undefined) symbol function, it's PC-relative (R_386_PC32) and targets the offset 22 of .text section, i.e., the 4 bytes representing the target of the call instruction.

Creation of the global symbol table. The linker collects all the symbol tables of the input object files and merges them. For instance, if an object file has an *undefined* symbol called “fibonacci”, and another object file *defines* a symbol with the same name, they will be merged, and the symbol will be assigned the address assigned to the section containing it plus the offset specified in the defining symbol. The linker also checks that each symbol is defined no more and no less than once.

Fix of the relocations. At this point the linker has all the necessary information to execute all the directives encoded in the object files: the relocations. The address of the symbol associated to the relocation is taken, encoded as requested (e.g., absolute address or relative to destination address) and written at the specified position.

The global symbol table is emitted in the final executable in the `.symtab` section, mostly for debugging purposes. In fact, just as the section list, after the linking process it is irrelevant. What really matters for execution purposes are *segments*. As mentioned, segments group sections with similar characteristics. More specifically, a segment represents a portion of the program which will be mapped in memory exactly as found in the file with a certain set of permissions. This is the reason why `.data` and `.bss` will end up in the same segment: they both require to be loaded in writeable, readable but not executable memory pages.

Typically, a program contains two segments: one for writeable data (such as the one just described) and one for executable (non-writeable) data. The latter segment will contain `.text` and `.rodata`. Note that, despite the fact that `.rodata` doesn’t need to be executable, it’s often mapped by the linker in an executable page. While it would be possible to map it in a read-only, non-executable page, this would introduce a cost in terms of memory.

The ELF format describes a segment in terms of a start offset in the file, the virtual address where it should be loaded, its permissions and the size it has in the file and the size it should have in memory. The size in the file and in memory can differ, due to the fact that the sections containing exclusively zeros (i.e., `.bss`) are not serialized into the file, but are just taken into account as extra size at the end of a segment.

When the program execution is requested, the operating system, or more specifically, the kernel, will parse the list of segments in the program and map the specified portions of the file in memory as requested, zeroing out the difference between the file segment size and the memory segment size.

1.2.3 Dynamic Loading

So far, we’ve been describing the process to produce a so called *static* binary, i.e., a program that uses no dynamic libraries, and whose code and data has to be loaded at a fixed address. In this section we will see how the concepts presented so far can be extended to support dynamic libraries, and the process to load them in memory at run-time: *dynamic loading*.

In ELF-based systems, dynamic libraries offer a series of crucial benefits: they allow to have a single copy of a set of functions used by multiple programs on the disk *and* in memory. In fact, dynamic libraries are designed in such a way that they can be loaded anywhere in the address space (i.e., not at a position determined at compile time) without changes. Code with this property is known as *position independent code* (PIC). Thanks to this feature, the operating system can have the same portion of the dynamic library file mapped multiple times, in different processes, at pages located at different addresses, but all mapped to the same physical page. This means that, despite the fact that a library, say the C standard library, can be used by hundreds of different applications at the same time, it will effectively be loaded in physical memory only once.

Therefore, a dynamic library, is similar to an executable file, except for the fact that all of its code is compiled as PIC, and the segment at the lowest address starts at address zero. The dynamic loader, a user space component (for Linux x86-64 usually located at `/lib64/ld-linux-x86-64.so.2`), will then load them where they fit best.

However, support for dynamic libraries is not limited to shared objects and PIC code. The linking process must be extended to allow the executable and dynamic libraries to have pending symbols even after the linking is finished. It will then be responsibility of the dynamic loader to fix them.

Therefore the concept of *visibility* of a symbol has been introduced. A symbol that has external visibility can be left pending at link time (as long as it is defined in a dynamic library specified on the linker command line), so that the dynamic loader will handle it.

The dynamic loader will complete the work left undone by the linker. In practice, this means that executables and dynamic libraries will have a dynamic symbol table (`.dynsym` section) and a list of dynamic relocations (`.rel.dyn` or `.rela.dyn` section). The main difference between normal relocations and dynamic relocations lies in the fact that dynamic relocations are not tied to a specific section (section are relevant exclusively in object files) but to a virtual address.

The dynamic loader will find all the information it needs in the `.dynamic` section, which contains the address and size of the dynamic symbol table and dynamic relocation table, plus the list of dynamic libraries the current executable/dynamic library depends on.

The problem at this point is: what are the targets of the relocations? Is is possible to have a relocation targeting an address in a read-only segment? The answer is no, in fact such a relocation would either trigger a segmentation fault or require the dynamic loader to temporarily map the containing page as writeable. Even ignoring the cost of such an operation, writing the page would trigger the copy-on-write mechanism of the operating system, which would create a new copy of the page, nullifying the above mentioned benefits of PIC code. To circumvent this problem, the GOT (Global Offset Table) and the PLT (Program Linkage Table) have been introduced.

The GOT. The Global Offset Table was introduced to handle accesses to global variables with external visibility (in most cases, defined in some dynamic library). The GOT is a table, stored in the `.got` section, containing an entry for each externally visible global variable. Each entry is the target of a dynamic relocation associated to the corresponding symbol.

```
$ objdump -d main
08048460 <main>:
...
 804847e: 89 04 24          mov    DWORD PTR [esp],eax
 8048481: e8 1a fe ff ff    call   80482a0 <function@plt>
 8048486: 8d 44 06 03       lea    eax,[esi+eax*1+0x3]
...

080482a0 <function@plt>:
 80482a0: ff 25 0c a0 04 08 jmp    DWORD PTR ds:0x804a00c
 80482a6: 68 00 00 00 00    push   0x0
 80482ab: e9 e0 ff ff ff    jmp    8048290 <_init+0x1c>

$ objdump -j .got.plt -s main
Contents of section .got.plt:
 804a000 3c9f0408 00000000 00000000 a6820408  <...............
 804a010 b6820408

$ readelf -r main
Relocation section '.rel.plt' at offset 0x264 contains 2 entries:
 Offset     Info    Type                Sym. Value  Symbol's Name
0804a00c  00000107 R_386_JUMP_SLOT        00000000   function
...
```

Listing 1.2: Example of PIC code employing the PLT. We can see that the call to function (which is defined in an external library) has been fixed by the linker to point to `function@plt`. The PLT stub jumps indirectly to the content of its `.got.plt` entry (`0x0804a00c`), which, as shown by `objdump -s` contains `0x080482a6` encoded in little endian. Therefore, the first time the PLT stub will be invoked, the indirect jump will proceed to the next instruction, which will push on the stack the identifier of the dynamic relocation 0 and invoke the dynamic loader. The dynamic loader will then resolve the relocation (targeting the `function .got.plt` entry at `0x0804a00c`) and from then on, the PLT stub will jump directly to `function` bypassing the dynamic loader.

Basically, instead of patching the instruction performing the memory access in `.text`, the code doesn't perform a direct access but first loads the address of the global variable from the corresponding GOT entry and then performs the load or store through the pointer in the GOT.

The net effect is that, at the cost of performing an extra memory access, the segment containing the code doesn't need to be patched.

The PLT. The method described above could work for calls to functions with external visibility as well. However, a program having externally visible symbols could take a long time to launch, since all the relocations would have to be resolved each time. This is particularly unsatisfying when functions that are never used during the regular execution of the program need to be resolved. The `abort` function, for example.

For this reason, the lazy loading mechanism has been introduced. Basically, a new GOT, employed exclusively for functions, is introduced and placed in the `.got.plt` section. In practice, when the program needs to call an external function, it performs a call to a small code stub in the Program Linkage Table (`.plt` section). The stub jumps to the address contained in the entry corresponding to the target function in the `.got.plt`. However, unlike the `.got`, relocations targeting `.got.plt` are not resolved at startup, instead their initial value is left. The initial value is a pointer to the instruction next to the above mentioned jump, which writes in a register the identifier of the relocation associated to the target function and then transfer control to the dynamic loader. The dynamic loader will resolve the requested relocation, and from then on, when the PLT stub is invoked, it will jump directly to the correct address.

Such a mechanism is known as *lazy loading*, since the relocation associated to a certain function is resolved the first time it is called, or it is never resolved, in case it's never called.

Listing 1.2 shows an example of PIC code employing the PLT.

1.3 LLVM

LLVM (formerly Low Level Virtual Machine) is an open source compiler framework which has gained popularity in recent years. The main reasons for its popularity lie in the fact that it is written in modern C++, well engineered, fast and simple to use. One of the most appreciated features is the simplicity and clearness, compared to its competitors, of its only intermediate representation: the LLVM IR. It is also very easy to write optimizations and analyses, known as *passes*, both for the mid-end and the back-end. Moreover, since its inception, LLVM was designed to be usable as a library, which allows, for instance, to merge the compilation and assembly phase (without ever having to serialize the assembly in textual form) and makes it particularly suitable to be employed as Just-In-Time (JIT) compiler. On top of this it has a great and active community and a less restrictive license compared to its main competitor, GCC.

The most popular LLVM front-end is `clang`, a front-end for C and all of its main derivative languages (including C++, Objective-C and OpenCL). Other front-ends are available, with different maturity degrees, for many other popular programming languages such as Rust, Fortran, Ruby, Java, Python and D.

Since in this work we will be working extensively with the LLVM IR, in the following we will provide a brief introduction to its main characteristics. Unless otherwise noted, we will be working with LLVM version 3.8.

1.3.1 The LLVM IR

The LLVM IR is an intermediate representation, typically emitted by a compiler front-end, suitable for being manipulated and optimized before begin passed on to the compiler back-end. In this section, we will briefly describe the LLVM modules, their structure and the LLVM type system.

The compiler works on a single translation unit. A translation unit in LLVM is represented by a *module*. A module is composed by a series of *global objects*, i.e., global variables and functions. A global object can be thought as a pointer to the storage associated to that object, that is either the code of the function or the actual data of the global variable. In fact, as we will see, global variables have pointer type.

Functions are basically a container of *basic blocks*. The order in which basic blocks appear in a function doesn't matter, except for the first one, which is elected as the *entry basic block*. This is due to the fact that each basic block has no *fallthrough* basic block, but each successor (if any) is explicitly specified.

In turn, basic blocks are composed by one or more *instructions*. Each basic block must be terminated by a *terminator instruction*, such as a branch (direct or indirect, conditional or unconditional), a switch instruction (analogous to the C `switch` statement) or an *unreachable* instruction, which has no successors and represents the fact that execution should never reach that point.

Before proceeding in the description of the other instruction types it is important to understand one of the key features of the LLVM IR: the SSA form. The Static Single Assignment form, is a form of IR where each variable can be assigned at most once. The concept of *variable* in the LLVM IR is translated into the concept of *value*. In practice most LLVM instructions produce a value (e.g., the result of an addition), implicitly creating a new *variable*. After the creation of the value, it can be only used (i.e., read) and never assigned again. The SSA representation of the code makes certain analyses or optimizations very easy. For instance, identifying an instruction whose result is never used is straightforward (its use list is empty).

In the following we list the LLVM instructions most relevant to our ends, many of which are exemplified in Listing 1.3.

Arithmetic/logic instructions. They represent the classic arithmetic/logic instructions. They typically have two operands and produce an output of the same type. Only the specified operation is performed, with no side effects.

`alloca`. Reserves a certain amount of space on the stack and returns a pointer to it. This the typical way in which front-ends emit local variables. Later passes, such as scalar replacement of aggregates (SROA), if possible, will try to promote them to SSA values. Apart from the above mentioned benefits, SSA values are also candidates for being assigned to registers during the register allocation phase.

`call`. A function call. It can have one or more arguments and a return value. It can be direct, when the target function is a known function, or indirect in case the destination is known at run-time only.

`load`/`store`. Given an address, they perform a load or store from memory. Global variables and local variables associated to an `alloca` instruction are read or written through memory stores.

`icmp`. Perform a comparison between two integers for equality, inequality, greater than and so on. Their result is a boolean (a 1-bit integer) which is typically employed as a condition in a conditional branch instruction.

`br`. The branch instruction can either be direct, if the destination address is explicitly identified in the code, or indirect, in case the target address is the result of some computation. Moreover a branch can either be unconditional or conditional. Conditional branches have three operands: two references to basic blocks and a single-bit integer which represents the condition based on which the former or the latter basic block should be executed next.

`switch`. Similar to the `br` instruction, it can have multiple successors. The taken successor is chosen depending on the value assumed at run-time of a certain SSA value. Similarly to the C `switch` statement, a list of possible constant values is listed, along with the corresponding target basic blocks.

`phi`. Phi nodes have an operand for each predecessor, representing the value to produce in case the current basic block is reached through that predecessor. This instruction is rarely emitted by front-ends, since its need typically emerges due to the promotion of a local variable to the SSA value status. Since, in this thesis, we focus on a front-end, we will rarely discuss it.

Another key component of the LLVM IR relevant for our purposes are *metadata*. A *metadata* is a piece of information that can be associated to an instruction and can carry any type of information: it's a general purpose tool to decorate an instruction with additional information. Each instruction can be associated to multiple *kinds* of metadata, each one of them identified by a name.

A metadata node can contain other metadata nodes, integers and strings. It is therefore possible to build sophisticated structures of information. Debug information are encoded in LLVM IR in a special form of metadata (the `!dbg` kind).

It is important to understand that optimization passes are not required to preserve metadata, they can more or less be freely dropped, debug information included.

```c
int function(int *);

int main(int argc, char *argv[]) {
  int result, local_variable = 3;

  if (argc > 3)
    result = 3 + argc + function(&local_variable);
  else
    result = 1;

  return result;
}
```

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

define i32 @main(i32, i8**) {
entry:
  %2 = alloca i32
  store i32 3, i32* %2, !dbg !22
  %3 = icmp sgt i32 %0, 3
  br i1 %3, label %true, label %epilogue

true:
  %4 = add i32 %0, 3
  %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
}

; ...
!22 = !DILocation(line: 4, column: 14, scope: !7)
```

Listing 1.3: Example of a simple C function and corresponding LLVM IR. `%2` corresponds to `local_variable`, which, unlike `result`, could not be promoted to SSA value since its address is passed to another function. `local_variable` is initialized to `3` by the `store` instruction. The `if` statement is implemented through an `icmp` instruction (where `%0` represents the first argument) and a conditional branch. In the `true` branch, the return value is computed through two additions and a function call. The call to `function` has as first and unique argument a pointer to `local_variable` (`%2`). Finally the value to return, `%7`, is chosen through a `phi` instruction: in case the `epilogue` basic block was reached from `true` the value `%6` is used, otherwise, if it was reached from `entry`, the constant `1` is employed. Note the `!dbg` metadata associated to the `store` instruction: it's a debug information describing the line, column and context from which it was generated.

Finally, it is important to understand how the LLVM type system works. Each global variable, function, SSA value and constant is associated to a type. The most relevant types available are:

IntegerType an integer type of a fixed size in bit.

PointerType a pointer to another type.

StructType a type composed by a sequence of other types.

ArrayType a type composed by a finite number of elements of a certain type.

FunctionType a type describing the prototype of a function (return value type and zero or more arguments). A function type can also be variadic, in which case it can have a statically unknown number of arguments after those explicitly listed.

1.4 QEMU

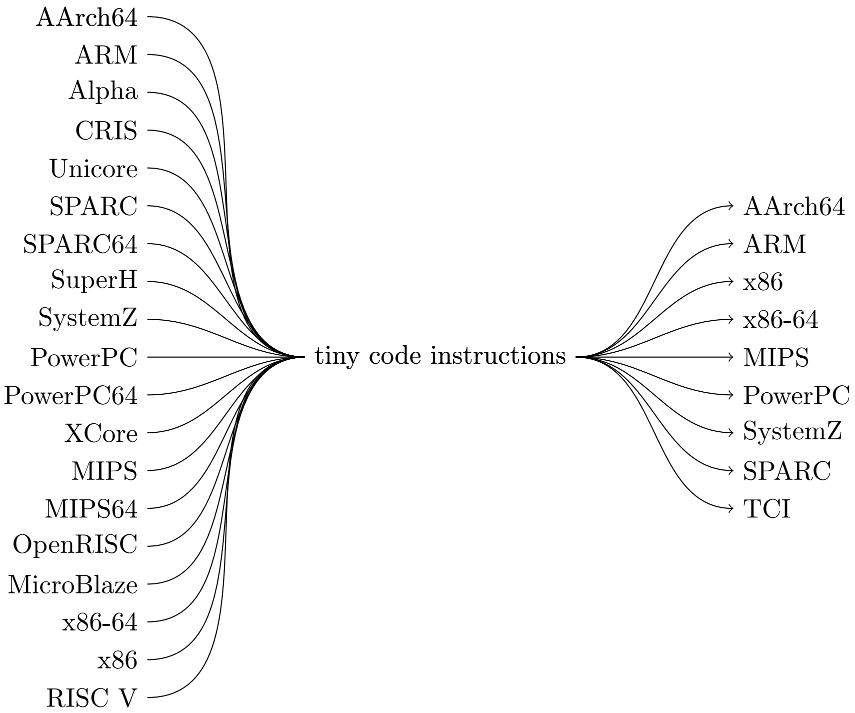


Figure 1.2: Architectures supported by QEMU. On the left, we have the front-ends, on the right, the back-ends.

QEMU is a *dynamic binary translator* (DBT). Given a program, it *emulates* it, one basic block at a time. Given an address, QEMU starts to decode the instructions at that address until a branch is found (i.e., the end of the basic

```
movi_i32 tmp0,$0x1005
movi_i32 tmp1,$0x4
sub_i32 tmp2,esp,tmp1
qemu_st_i32 tmp0,tmp2,leul,0
mov_i32 esp,tmp2
movi_i32 eip,$0x2000
```

Listing 1.4: Tiny code instructions generated from a x86 `call` instruction. The `call` instruction is located at address `0x1000`, it's 5 bytes long (therefore the return address is at `0x1005`) and it's performing a call to a function at address `0x2000`. The return address is first materialized into `tmp0`, the constant 4 is put in `tmp1`, then the difference between the current value of `esp` and `tmp1` is computed and put in `tmp2`. Then, the value of `tmp0` is stored at the address represented by `tmp2` and the value of `esp` is then updated with `tmp2`. This code is basically pushing the return address on the top of the stack. Finally, the program counter `eip` is updated with the address of the target function.

block). Then the instructions are translated from the input architecture to a set of equivalent instructions of the host architecture. For instance, if the user wants to run an ARM program on a x86 machine, the ARM instructions will be translated one by one in a set of equivalent x86 instructions, which will be then executed.

What is interesting in the QEMU design is the fact that it decouples the process of interpreting what a certain input instruction does from emitting the equivalent code using instructions of the host architecture. In practice, this means that, just as compilers do, QEMU doesn't directly translate an input instruction into equivalent instructions for the host, but first goes through an intermediate representation.

The intermediate representation is composed by *tiny code instructions*, and the component taking care of transforming them into code that the host can run is the *tiny code generator*.

In this sense, as shown in Figure 1.2, QEMU has a *front-end*, the part interpreting input instructions and emitting tiny code instructions, and a *back-end*, the part emitting instructions for the target architecture. This decoupling is particularly beneficial, since if we want to introduce a new input (target) architecture, it is enough to implement a front-end (back-end) and it will be automatically able to operate with all the existing back-ends (front-ends).

In Listing 1.4 a snippet of the tiny code instructions generated due to an x86 `call` instruction is presented. The code is similar to the intermediate representation of a compiler. Instructions have no side effects, therefore, the behavior of the input instruction is made completely explicit. Note also how QEMU employs temporary variables (`tmp#`) for temporary computations and global variables to represent the CPU state (e.g., the `esp` and `eip` registers)

Another key feature of QEMU are its two modes of operation:

System mode. Full hardware emulation. Suitable for booting a whole operating system or running bare metal code. Peripherals, MMU and all other hardware components are emulated.

User mode. Emulation of the operating system only. This mode of operation allows, e.g., ARM binaries for Linux to be run on an x86 Linux host. Only the interfaces with the operating system are emulated, such as syscalls, signals and so on. This is possible thanks to a layer, that we call the *QEMU Linux subsystem* that forwards, with the appropriate changes, syscall arguments for the emulated system to the host system. In practice, an `open` syscall will be forwarded as is to the host operating system.

The *user mode* will be of greatest interest for us. In practice, the user mode allows Linux binaries to be run on Linux installations for a different architecture. The same is possible for FreeBSD. However, support for it it's currently broken in QEMU.

In terms of tiny code instructions syscalls are handled by calling an external function, written in C, which will perform the necessary operations. Such functions are known as *helper functions*. *Helper functions* are not limited to handling interactions with the operating system, they are also employed to translate instructions that would be too complicated to be implemented in terms of tiny code instructions. An example is the floating point division.

1.5 Monotone Frameworks

A Monotone Framework is a way to define a data-flow analysis in a general and effective way. A data-flow analysis associates a *value* to the entry and exit of each program *label*, typically, an instruction or a basic block. The value at the exit of a *label* is the value at the entry after the effects of the code in the *label* have been applied. These effects are different depending on the content of the *label* and are modeled by a *transfer function*. On the other hand, the value at the entry of a *label* is obtained by *combining* all the values at the exit of the predecessors' labels.

As an example, consider the *even-odd analysis*, which tries to track if the value of a certain variable, in a certain program point, is even or odd. We will have three possible *values*: EVEN, ODD or ANY. The initial state will be ANY. It's easy to formulate a transfer function by laying out some simple rules: 1) if the variable is assigned to an even (odd) constant, we move to the EVEN (ODD) state, 2) if we multiply the variable by 2, we go to EVEN, 2) if currently the variable is in EVEN (ODD) state and we add 1 we move to ODD (EVEN). It's also easy to define how values should be *combined*: suppose we have an `if` statement where the value after the true branch is EVEN, and after the false-branch is ODD, the common successor of the two branches will have to take into account both possibilities, producing the state ANY. If, on the other hand, both branches were to give the same result, say EVEN, then the common successor would be associated to EVEN too. Figure 1.3 illustrates this example. From the figure, we can derive the following set of equations:

$$n_\bullet = f_n(n_\circ)$$

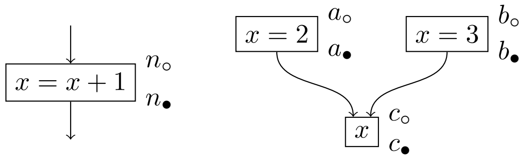


Figure 1.3: Graphs representing two simple programs. Each box represents a basic block.

$$\begin{aligned} a_\bullet &= f_a\,(a_\circ) \\ b_\bullet &= f_b\,(b_\circ) \\ c_\circ &= a_\bullet \sqcup b_\bullet \end{aligned}$$

where f_ℓ represents the *transfer function* of label ℓ, $\ell_\circ$ the data-flow value at the entry of label ℓ, and $\ell_\bullet$ the value at its exit. $\sqcup$ is the operator that *combines* two data-flow values, producing a new value which is *more generic* than the two input ones. In this case f_n would be defined as producing ODD if its argument is EVEN, EVEN if its argument is ODD or ANY if its argument is ANY. On the other hand, f_a and f_b would produce ODD and EVEN respectively, no matter what their argument is.

Therefore, supposing $n_\circ = \text{Odd}$, a valid assignment for the variables of the set of equations above would be: $n_\bullet = \text{Even}$, $a_\bullet = \text{Even}$, $b_\bullet = \text{Odd}$ and $c_\circ = \text{Any}$. Such a set of assignments are said to be *solutions* of the data-flow equations.

The presented analysis can track three possible different values. Note however that the set of possible different values in a data-flow analysis can be infinite, but they have to be part of a partial ordering relation. The partial ordering needs to have a top ($\top$) element and a bottom ($\bot$) element. The ordering between *elements* of the lattice, denoted as $a \sqsubseteq b$, can be read as "b is more generic than a", "b is less informative than a", or "b *contains* a". In our previous example, ANY is more generic than ODD. If ANY, ODD and EVEN are thought in terms of set of natural numbers, then ANY also *contains* ODD: $\{1, 3, 5, 7, \ldots\} \subseteq \{1, 2, 3, 4, 5, \ldots\}$. On the other hand, ODD is more *informative*, in the sense that it can tell us something more specific about the value of the variable we are tracking, compared to ANY. In this specific case, therefore, ANY represents $\top$.

Consider the set of assignments of variables of the data-flow equations composed exclusively by $\top$. It would be correct (supposing we replace $=$ with $\sqsubseteq$), but very little informative, or, to be more precise, it would provide no information at all. The idea of the data-flow analysis is to obtain the most informative solution for the equations, without compromising its correctness.

A more formal definition. The data-flow information is represented by the *property space* L, i.e., the set of all the possible *values* that we can associate to a label. In a Monotone Framework, L must be combined with a *partial ordering operator* $\sqsubseteq: L \times L \to \{true, false\}$, i.e., $\sqsubseteq$ is reflexive, anti-symmetric and transitive.

We call $(L, \sqsubseteq)$ a *partially ordered set.*

In a Monotone Framework, the *partially ordered set* $(L, \sqsubseteq)$ must have finite height. This means that all subsets Y of L where all the elements can be compared according to $\sqsubseteq$, must be finite. Or, in symbols: $\forall Y \subseteq L : \forall l, l' \in Y, (l \sqsubseteq l') \vee (l' \sqsubseteq l)$, Y has a finite number of elements. This is the so called *Ascending Chain Conditions* [105].

$(L, \sqsubseteq)$ must also form a *complete lattice.* To define the concept of *lattice* we have to introduce the concepts of *upper bound* and *least upper bound.* Given a subset Y of L, an element $l \in L$ is said to be an *upper bound of* Y if it is greater than any other element $l' \in Y$, or in symbols, $\forall l' \in Y$, $l' \sqsubseteq l$. On the other hand, if l' is an *upper bound of* Y and it is lower than or equal to all the other upper bound elements of Y, then it's said to be the *least upper bound of* Y. If a subset Y has a *least upper bound*, then it's unique and it can be denoted as $\bigsqcup Y$. The concepts of *lower bound* and *greatest lower bound* can be easily defined by duality. A *complete lattice* is a partially ordered set $(L, \sqsubseteq)$ such that all subsets of L have least upper bounds and greatest lower bounds. A complete lattice is therefore defined by the following elements:

L a set of elements.

$\sqsubseteq$ the partial ordering operator.

$\bigsqcup$ the *least upper bound* operator.

$\bigsqcap$ the *greatest lower bound* operator.

$\top$ the top element $\top = \bigsqcup L$.

$\bot$ the bottom element $\bot = \bigsqcup \varnothing$.

If we are computing the least upper bound of two elements $\bigsqcup \{l_1, l_2\}$, then we can also use the combine operator $\sqcup : L \times L \rightarrow L$ defined as $l_1 \sqcup l_2 = \bigsqcup \{l_1, l_2\}$.

The data-flow equations can be therefore be defined as follows:

$$\text{Analysis}_\circ(\ell) = \begin{cases} i & \text{if } \ell \in E \\ \bigsqcup \{\text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} & \text{otherwise} \end{cases}$$

$$\text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell))$$

where:

Analysis$_\circ$ (ℓ) represents the value (solution) associated to the entry of label ℓ.

Analysis$_\bullet$ (ℓ) represents the value (solution) associated to the exit of label ℓ.

E represents the *extremal labels*, i.e., the set of entry point labels.

i represents the *extremal value*, i.e., the value associated with the entry point(s) of the program E.

$\bigsqcup$ is the *least upper bound operator*, as defined before.

F represents the set of edges of the control flow in the form (ℓ', ℓ) where ℓ' is the source of the edge and ℓ the destination.

$f_\ell \in \mathcal{F}$ is the transfer function for the label ℓ, where $\mathcal{F}$ is the set of all the transfer functions. Note the transfer functions have to be monotone [105].

The MFP solution. Once all the equations have been set out, we need to obtain a (non-trivial) solution to them. The Maximal Fixed Point (MFP) solution is the most popular approach to obtain a solution for a Monotone Frameworks since it provides good solutions and termination guarantees.

Algorithm 1.1 presents the MFP solution for Monotone Frameworks. The algorithm works on a queue ($Worklist$), initialized with all the edges $(\ell, \ell') \in F$ of the control-flow graph. The idea of MFP is to start from a base solution which is then iteratively refined. The temporary result is stored in an associative array *tmp* mapping labels to elements of L. *tmp* holds the data-flow information at the beginning of the label (analogously to $\text{Analysis}_\circ$) and is initialized with i for all the entry labels $\ell \in E$ and with $\bot$ for all the others.

After the initialization phase, the algorithm pops an edge (ℓ, ℓ') from $Worklist$, takes the last result produced for ℓ (i.e., $tmp\,[\ell]$) and applies to it the transfer function for the label ℓ (f_ℓ). If the result is lower than or equal to the last result produced by the successor of ℓ currently being considered $tmp\,[\ell']$, then we are not producing any new information. Otherwise, we update the last result for ℓ' merging into it the result of the transfer function ($f_\ell\,(tmp\,[\ell])$). Then, since we changed the information associated to ℓ', we re-enqueue all of its successors ℓ'' such that $(\ell', \ell'') \in F$.

Once all of this process is finished, we emit the final solution as $MFP_\circ$ (the solution at entry of a label) and $MFP_\bullet$ (the solution at exit of a label). The former is simply a copy of *tmp*, while the latter is *tmp* after applying to each element $tmp\,[\ell]$ the respective transfer function f_ℓ.

While proving the termination of the algorithm is outside the scope of this work, it's interesting to get an intuition of why it always terminates. The only possible source of non-termination relies in the possibility of re-enqueuing an edge an infinite number of times. Due to the algorithm structure, an edge can be re-enqueued only if the temporary result for the successor $tmp\,[\ell']$ does not contain the result at the end of the current label $f_\ell\,(tmp\,[\ell'])$. If this happens the temporary result of the successor is updated to include $f_\ell\,(tmp\,[\ell'])$. However, since at each iteration $tmp\,[\ell']$ will move closer to $\top$, and since the lattice is guaranteed to have a finite height (thanks to the *Ascending Chains Condition*), this can happen only a limited number of times. The termination of the algorithm is therefore guaranteed.

It is also interesting to note that the presence of $\bot$ in the lattice is required only to initialize the values associated to non-entry labels, so that they will absorb directly whatever value is propagated first by their predecessors (since $\forall l \in L$, $l \sqcup \bot = l$). Apart from this, having a bottom element is not required, and, in fact, in our *even-odd analysis* we didn't have it. In such cases, the $\bot$ element can be artificially introduced in the lattice.

An alternative approach popular to build a data-flow analysis is *abstract interpretation*. For information about its relationship with Monotone Frameworks, see [147].

```
Data: A Monotone Framework $(L, \mathcal{F}, F, E, i, f)$
Result: $MFP_\circ$, $MFP_\bullet$
/* Initialization                                        */
$Worklist = \{\}$;
$tmp = \{\}$;
foreach $(\ell, \ell') \in F$ do
    $Worklist.enqueue((\ell, \ell'))$;
end
foreach $\ell \in E$ do
    $tmp[\ell] = i$;
end
foreach $\ell \in F \backslash E$ do
    $tmp[\ell] = \bot$;
end
/* Iterative refinement of the solution                  */
while $not\ Worklist.empty()$ do
    $(\ell, \ell') = Worklist.pop()$;
    if $f_\ell(tmp[\ell]) \not\sqsubseteq tmp[\ell']$ then
        $tmp[\ell'] = tmp[\ell'] \sqcup f_\ell(tmp[\ell])$;
        foreach $\ell''$ such that $\exists(\ell', \ell'') \in F$ do
            $Worklist.enqueue((\ell', \ell''))$;
        end
    end
end
/* Finalization                                          */
foreach $\ell \in F$ do
    $MFP_\circ[\ell] = tmp[\ell]$;
    $MFP_\bullet[\ell] = f_\ell(tmp[\ell])$;
end
```

Algorithm 1.1: The algorithm to obtain the Maximum Fixed Point solution for a data-flow problem.

Chapter 2

A rev.ng Overview

rev.ng is a binary analysis framework based on QEMU and LLVM at its core. Its primary goal is building a framework to perform static and dynamic binary analysis that can work on a large set of diverse architectures. rev.ng also aims at recovering high quality abstractions and offering a representation of the program accurate enough to be recompiled and preserve the original behavior. The final grand goal is to build abstractions sophisticated enough to build a *unified decompiler* producing high-quality, recompilable C code.

It is mainly composed of about C++ 17 kSLOC and it's publicly available under a Free Software license [48]. Its main goal is to combine the benefits of QEMU, a stable emulator with support for a large range of architectures, with those of LLVM, a mature compiler framework highly suitable to perform any sort of analysis and recompile the produced code.

It is important to understand that, despite the fact that it makes large use of QEMU, rev.ng is mainly a static binary analysis tool: the code is never run during the analysis process. As explained in Section 1.4, QEMU is similar in structure to a modern compiler: it features a *front-end* that parses binary code and emits an *intermediate representation* of the input basic block, and a *back-end* that compiles the intermediate representation into machine code for the host architecture. rev.ng exploits exclusively QEMU's front-end.

Then, starting from QEMU's IR, rev.ng translates it to equivalent LLVM IR, on top of which a series of analyses are performed. Once all the code has been translated, since LLVM is a compiler framework, it's straightforward to recompile it and produce a binary with the same behavior of the original program. The ability to recompile the code gives birth to the core component of rev.ng: the *static binary translator*.

2.1 Requirements and design criteria

As previously mentioned, one of the key goals of rev.ng is to be able to handle multiple architectures in a unified way. This has two implications: 1) rev.ng must have a way to obtain a uniform representation of the behavior of each instruction from its raw machine code, and 2) all the further operations performed on this representation must not make assumptions about the original architecture.

The former challenge is solved by employing QEMU, while the latter affects the design of our analyses. For instance, in rev.ng, initially, we lack apparently basic knowledge such as what a *function call* or a *return instruction* are. Therefore, we will have to redefine these concepts in an architecture-agnostic way and write a small analysis to detect them. Similarly, when we will describe the anlaysis to identify function arguments and return values, it won't only be architecture-independent, but also ABI-indepedent.

The general idea behind this approach consists in focusing the analysis efforts on the *features* of an ISA, and not on the ISA itself. For example, in Section 5.2 we will describe how to handle ARM predicate instructions, but the techniques we will explain will work as good in case of conditional moves in x86-64 and other architectures.

While this approach introduces a sensible engineering cost, in the long term it will be beneficial, since the marginal cost for supporting a new architecture will decrease over time.

Another key requirement, along with being platform independent, is to develop analyses that are semantics-preserving. Often times, binary analysis tools provide information that is useful for an analyst but not accurate enough to guarantee correctness in case of recompilation. In fact, many tools (such as the IDA Pro decompilers) produce code that is not even thought to be recompiled.

rev.ng aims at addressing this issue in two ways. First, it provides to the user an abstraction-poor representation which is guaranteed to be very close to the original program. Then, if the user is interested in recovering higher level abstractions such as function boundaries, function arguments and so on, rev.ng trades some correctness for the sake of providing useful and accurate information in most of the cases, but, at the same time, it features run-time fallback paths in case the results of an analysis was mistaken.

For instance, in Section 4.2.2 we will explain how we can identify function boundaries: this information will be employed to isolate the corresponding basic blocks into functions. In case our algorithm misidentifies a function, we foresee a fallback mechanism which, at run-time, will lead program execution to a slower, but safe, path.

2.2 revamb: a Static Binary Translator

A *static binary translator* is a tool that, given an input program for a certain architecture, produces a program with an equivalent behavior for another architecture (or even the same). The core component of rev.ng is its static binary translator, revamb. Figure 2.1 offers an overview of its translation process.

revamb takes as input a statically linked Linux program, goes through its segments (see Section 1.2.2) and identifies those containing data and those containing code. It employs the appropriate QEMU's front-end to translate into tiny code instructions each basic block in the input program. Then the tiny code instructions are translated into LLVM IR and collected in a recompilable LLVM module.

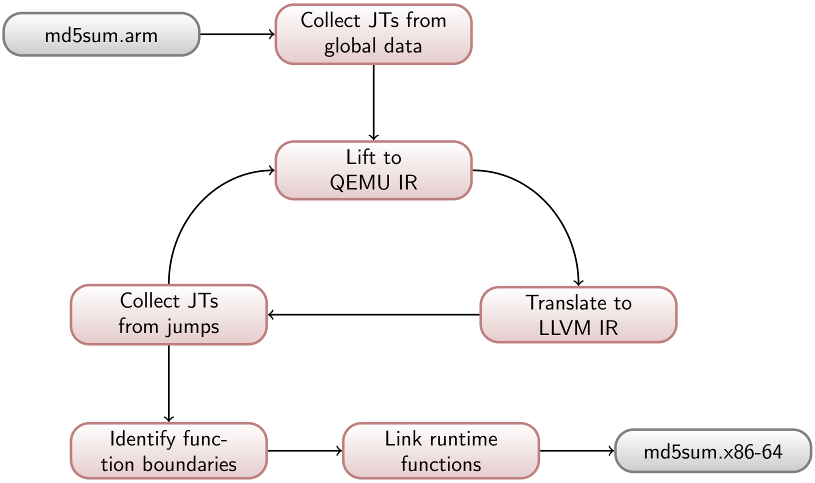


Figure 2.1: Overview of the static binary translation process. The input binary is initially fed to the pre-processing phase which collects an initial set of *jump targets* from the global data and the program's entry points. Then the iterative discovery of *jump targets* begins: the code at the address corresponding to each jump target is first translated into *tiny code instructions* and into LLVM IR, which is then analyzed to recover additional *jump targets*. When no new *jump targets* can be found, further analyses are performed (such as the identification, see Chapter 4). Finally, the program is linked against the necessary libraries and the final executable is emitted.

The final result is a new binary where those segments are located at their exact original location, so that all the load and store instructions targeting an absolute address (typically, a global variable) can stay as they are. In the LLVM module, each input segment is represented as a global variable associated to a section whose address will be forced, at link time, to be the address the segment had in the original program.

This isn't true for data segments only, in fact, also the segment containing the executable code can contain data (such as the `.rodata` section or constant pools). While the original code and data will remain where they were, the translated code (i.e., the code that will actually be executed) will reside in another location of the address space.

This approach allows us to avoid performing instrumentation of memory accesses and the associated overhead. In practice, the address space of the translated program will be, at least initially, exactly the same as the original binary, except for the fact that there will be an extra segment containing the translated code.

Once the layout of the address space has been fixed, the translation has to start. The translation process presents four key challenges: how the state of the CPU should be represented, how to identify basic blocks, how to link them to each other and how to handle the interactions with the operating system (e.g., syscalls). In the following we will present these challenges. This section will then end with the description of the simplest analysis that can be performed on the recovered LLVM IR: function call detection.

2.2.1 Representation of the CPU State

In `revamb` the CPU state is represented in a way analogous to how QEMU represents it (see Section 1.4). In fact, both QEMU and `revamb` represent each individual portion of the CPU state through a global variable. In practice, each register or CPU flag will have a corresponding global variable in the LLVM IR. We call such a global variable *CPU State Variable*, or simply CSV. This means that every instruction reading or writing a register will perform a *load* or a *store* to the corresponding global variable. For this reason, while it might sound counterintuitive at the beginning, we will often talk about performing loads and stores to registers.

Representing CPU registers, whose key characteristic is to have very low access latencies, with global variables, which reside in memory and are notoriously hard to optimize, can lead to a large drop in terms of performance. However, since registers *are* a form of global state, this is a problem inherent to binary translation that, e.g., QEMU has too. Note however that, depending on the cleaness of the control-flow graph, the LLVM optimizer will be able to elide many store/load pairs. A more radical solution would consists in recovering enough abstractions to identify functions and their arguments, promoving register arguments to actual formal arguments of LLVM functions (a topic tackled in Chapter 4 and Chapter 5).

Another option would be to try to pin general purpose registers of the input architecture to a set of general purpose registers of the target architecture. However this would not always possible in case of mismatches in the number of available general purpose registers between the two architectures. Moreover, it would seriously hinder the portability of the code and reduce drastically the set of registers available to the register allocator, making the integration with the LLVM framework very hard. Therefore we don't deem such a path promising, and prefer to focus our efforts in retrieving abstractions leading us to produce a code more similar to that produced by a classic LLVM frontend (e.g., clang) and enabling it to benefit from more general purpose optimizations.

2.2.2 Basic Block Identification

While it might sound straightforward at first, the exhaustive enumeration of all the basic blocks in a program is a challenging task. In fact, any address with an appropriate alignment within an executable segment is the potential start of a basic block. To make such a situation more clear, the reader can think about the situation where an attacker can obtain direct control of the program counter: he will be able to jump at any address in an executable memory page. Therefore, the most conservative solution would be to mark each address with an appropriate alignment within an executable segment as the starting point of a basic block. However, this would seriously affect the performance of the program in case of recompilation and make any further analysis extremely difficult, if possible at all. To understand why it's sufficient to imagine the control flow graph of a program where each indirect jump can reach any other address within the program: any optimization would be inhibited and even just walking the control flow graph backwards (from a basic block through its predecessors) becomes a confusing task.

In conclusion, the identification of basic blocks has two conflicting goals: 1) identify all of the legitimate basic blocks and 2) produce a CFG as clean as possible to enable optimizations and meaningful analyses.

In `revamb`, the identification of the starting addresses of basic blocks (also known as *jump targets*) proceeds in an iterative way. A first set of jump targets is obtained from the image metadata, such as the entry point and exported functions. Then, all the segments are scanned looking for pointer-sized values that look like addresses pointing within an executable segment. This step allows us to capture all the global variables initialized with function pointers, C++ virtual tables and jump tables generated due to `switch` statements. This initial set of addresses are fed to QEMU which will produce its own intermediate representation, that, in turn, will be translated into LLVM IR. At this point the generated code is analyzed to discover new jump targets in two ways: 1) by collecting all the targets of direct and, where feasible, indirect branch instructions and 2) by collecting literal constants materialized in the code which appear to be a jump target. The latter approach allows to catch pointers to functions which are not stored in global variables, virtual tables or jump tables, but are hardcoded in the executable code itself.

The described method to identify basic blocks is a good compromise between marking all the addresses as jump targets (fully conservative, little informative) and obtaining an exact representation of the legitimate control-flow graph (very informative, hard if not impossible to obtain in the general case).

The obvious disadvantage of this approach consists in what we call *overtranslation*: it might happen that we mistakenly consider a piece of data as code. A typical situation is when, while scanning through global data, we find a pointer to a piece of data in `.rodata`. In fact, as we saw in Section 1.2.2, `.rodata` is often placed in the executable segment. Therefore, despite being just data, it is actually executable, and there's no 100% reliable way to detect such a situation. An even trickier case is represented by constant pools, which typically are placed between actual functions.

Note that *overtranslation* is a non-issue in terms of preserving the semantics of the program, since we're simply translating additional code which will never be run. However, it might be problematic in case further analyses are to be run on the code.

Chapter 3 will eviscerate this topic in depth.

2.2.3 Organization of the Translated Code

Each input basic block is translated into one or more LLVM basic blocks. In fact, even a single instruction can be expanded to multiple basic blocks. This is due to the fact that in the QEMU IR (and therefore in LLVM) each instruction performs a single operation and has no side effects. Therefore, instructions such as conditional moves or predicated instructions need to be expanded into the equivalent of an `if` statement.

All these basic blocks are collected into a single large LLVM function, known as *root*. Control flow transfer from one basic block to another one is performed in

```
%0 = load i32, i32* @pc
switch i32 %0, label %abort [
  i32 0x10074, label %bb.0x10074
  i32 0x10080, label %bb.0x10080
  i32 0x10084, label %bb.0x10084
  ...
]
```

Listing 2.1: Example of the dispatcher. The value of the program counter is read and fed to a very large `switch` statement which maps each *jump target* address to the corresponding LLVM basic block.

two ways: 1) directly in case the basic block is terminated by either a direct branch or an indirect branch for which we were able to statically exhaustively enumerate all the possible destinations, or 2) going through the *dispatcher*. The *dispatcher* (Listing 2.1) role is to route program execution to the correct basic block at run-time in case we weren't able to determine where it should statically. In practice it's a portion of the *root* function that compares the current (run-time) value of the program counter against all the addresses of the basic blocks that have been translated, and *dispatches* execution to the correct one.

Note that, as mentioned before, the lesser the dispatcher is employed, the better. Both for analysis purposes and performance of the recompile code. In fact, the switch statement is very large and sparse, therefore even its most efficient implementation will have a cost that is either logarithmic in time (binary search) or linear in space (hash table). Therefore it's critical to be able to statically enumerate all the targets of indirect branches local to a function. Ideally, the dispatcher should be used exclusively for indirect function calls.

2.2.4 Handling of Operating System Interactions

Being able to correctly translate single basic blocks and link them together is critical to be able to successfully produce a program with the same behavior as the original one, but it's not enough. Interactions with operating system have to be handled too. Specifically, process initialization, syscalls and signals have to be managed.

Process initialization. A Linux program starts its execution in an address space where its image has been loaded and the stack is initialized with a specific structure, mainly arguments and environment variables. In the case of the translated program, while the operating system will do the above mentioned operations, we need to set up an additional stack. In fact, we will have two stacks: one used to emulate the stack of the original program, and another one used for the hosting program. For example, if we are running on an x86-64 host a program originally built for ARM, the global variable representing the ARM `sp` register will point to the former stack, while the actual x86-64 `rsp` register will point to the latter. These two stacks evolve in distinct ways.

While the host's stack is initialized by the operating system, we have to take care of allocating and correctly initializing the emulated stack. This means copying the environment variables, program arguments, auxiliary vectors and so on the emulated stack. Once this operation has been performed, the *root* function can be invoked and the translated code can start to run.

Syscalls. Instructions that perform syscalls are handled by QEMU by emitting a call to an *helper* function (see Section 1.4). These functions are implemented in C and take care of translating the system call's calling convention from the one of the input architecture to the one of the host architecture. In practice syscalls are, after some processing, *forwarded* to the host operating system.

In `revamb`, we extract the necessary helper functions from QEMU, what we call the *QEMU Linux subsystem* (Section 1.4), and we link it into the output binary. The result is a self-contained binary which automatically forwards syscalls to the hosting operating system. Note that this is necessary only when translating a program from an architecture to a different architecture. In case the source and target architectures match, it is in principle possible to perform the syscalls unmodified.

Signals. Despite not requiring very sophisticated handling, signals are not currently handled by `revamb`.

2.2.5 Identification of Function Calls

In this section we will present a key analysis to understand the approach we follow in `rev.ng`, and its potential. In `rev.ng`, every time we write an analysis, we work on an LLVM module, we don't care what was the original architecture: all the analyses have to be architecture independent. This is a key principle to ensure that, once a new front-end is plugged in, all the analyses still work and produce results of quality close to architectures that have been supported for a long-time.

The fact that the LLVM module we work on is architecture independent also means that the code provides very little abstractions. This is true up to the point that even a very simple and common instruction such as `call` (or `bl` in ARM, or `jal` in MIPS) are not explicit in the generated code.

Therefore, the need to have an analysis to identify function calls arises. The *function identification analysis* is as simple as important, since we'll rely on its results several times in the rest of this work.

Before describing the implementation of the analysis, we need to provide a definition for *function call*:

Function call. A branch instruction which is preceded by an instruction storing, either in a register (the *link register*) or on the top of the stack, the *return address*. The *return address* is a constant integer whose value corresponds to the address of the current instruction, plus its size.

Given this definition, it's straightforward to write an analysis which goes through the code and checks each branch instruction (direct or indirect) to see if the immediately preceding instructions are saving the return address on the stack or in a register.

If such a situation is identified then a marker is emitted right before the branch instruction, representing the fact that the following branch is actually a function

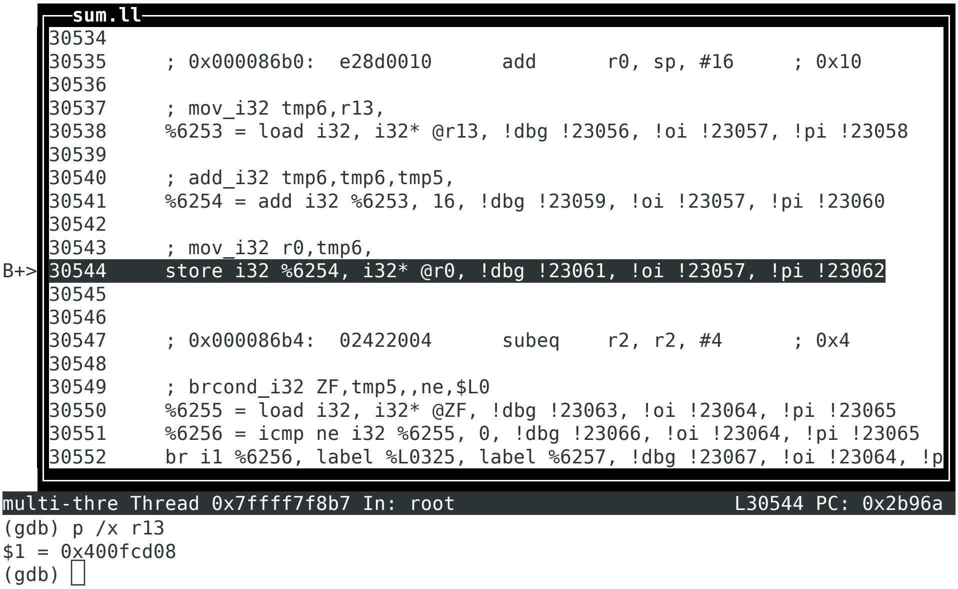
```
define void @root(i64) {
bb.main:
  ; 0x4006cf: call 0x4004f0
  %219 = load i64, i64* @rsp
  %220 = sub i64 %219, 8
  %221 = inttoptr i64 %220 to i64*
  store i64 0x4006d4, i64* %221
  store i64 %220, i64* @rsp
  store i64 0x4004f0, i64* @pc
  call void @function_call(i8* blockaddress(@root, %bb._init.0x8),
                           i8* blockaddress(@root, %bb.main.0x36),
                           i32 4196052)
  br label %bb._init.0x8
; ...
}
```

Listing 2.2: Code generated by a x86-64 `call` instruction. The first five instructions are pushing the return address on the stack. In fact `0x4006d4` is the address of the call instruction plus its size, and its being stored on the top of the stack (after it has been decreased by 8). Then the program counter is updated and we can see a branch to the target function. The `call` to `function_call` is the marker. It's not a real function call, its only purpose is marking that the next instruction is a function call. Its first argument represents the entry basic block of the called function. The second argument is the return basic block, while the third argument represents its address.

call. The marker also encodes what is the called function (basic block), what's the return basic block and its address. An example of the results of the function call detection is shown in Listing 2.2.

2.3 Advanced Features

This section introduces a series of advanced features, with respect to the basic functionalities presented so far. Specifically, we will first provide an overview of the debugging capabilities embedded in `rev.ng` to make the diagnosis of mismatches between the behavior of the translated and original program easier. We will then present the challenges introduced by dynamic libraries and how we intend to tackle them accurately. Afterwards, this chapter provides a walk-through on how to implement a simple static instrumentation using our homebred LLVM Python bindings, `llvmcpy`. Finally, we will illustrate how we intend to extend the platforms handled by `rev.ng`, both beyond the set of architectures supported by QEMU (by employing CGEN, a simulator/disassembler generator) and in terms of operating systems and binary formats.



```
sum.ll

      ; 0x000086b0:  e28d0010      add      r0, sp, #16     ; 0x10

      ; mov_i32 tmp6,r13,
      %6253 = load i32, i32* @r13, !dbg !23056, !oi !23057, !pi !23058

      ; add_i32 tmp6,tmp6,tmp5,
      %6254 = add i32 %6253, 16, !dbg !23059, !oi !23057, !pi !23060

      ; mov_i32 r0,tmp6,
B+>   store i32 %6254, i32* @r0, !dbg !23061, !oi !23057, !pi !23062


      ; 0x000086b4:  02422004      subeq    r2, r2, #4      ; 0x4

      ; brcond_i32 ZF,tmp5,,ne,$L0
      %6255 = load i32, i32* @ZF, !dbg !23063, !oi !23064, !pi !23065
      %6256 = icmp ne i32 %6255, 0, !dbg !23066, !oi !23064, !pi !23065
      br i1 %6256, label %L0325, label %6257, !dbg !23067, !oi !23064, !p

multi-thre Thread 0x7ffff7f8b7 In: root                    L30544 PC: 0x2b96a
(gdb) p /x r13
$1 = 0x400fcd08
(gdb) 
```

Figure 2.2: Screenshot of gdb stepping through the generated LLVM IR. The highlighted line represents the store writing the result of the add instruction (add r0,sp,#16) into r0. In the gdb shell, the value of the r13 register is being printed on screen. We take huge pride in the fact that this figure, despite being a screenshot, is fully vectorial. No vectorial images were harmed in the making of this thesis.

2.3.1 Debugging

Due to programming errors in rev.ng or violations of some assumptions we made, it might happen that a translated program misbehaves, i.e., it doesn't behave as the original program. In such cases, tracing back the source of the problem can be a challenging task, since setting a register to a wrong value could go unnoticed for most of the program execution, until the problem shows up. This problem is exacerbated by the fact that, if we consider the original C program being translated, there are four translation layers between the source code and the running assembly: 1) compilation from C to the target assembly, 2) translation from assembly to QEMU's tiny code instructions, 3) translation from QEMU's tiny code instructions to LLVM IR, and 4) translation from LLVM IR to the target architecture.

For this reason, in the LLVM module each instruction is associated to debugging information (see Section 1.3) which allows the developer to trace a certain output instruction back to the corresponding LLVM IR instruction, or tiny code instruction or assembly instruction, at user's choice. In LLVM, this is implemented through !dbg metadata, which allows to encode debugging information in a portable way: the compiler will then take care of emitting them in the appropriate format depending on the target image format (ELF, in this case).

As shown in Figure 2.2, the end result is the possibility of employing a debugger (such as gdb) to step through the generated LLVM IR, tiny code instructions or the original assembly. Moreover, since the CPU state is represented as global variables, inspecting the value of a register is straightforward.

2.3.2 Dynamic Libraries Support

So far we've been discussing how statically linked binaries are handled by revamb. But what about dynamically linked programs? The handling of dynamically linked programs can be tackled in two different ways.

A first option is to simply load all the libraries individually before starting the analysis, and then treat the combination of the main program and libraries as if it was a statically linked binary. This is the most straightforward approach, except for the handling of dynamic relocations (see Section 1.2.3), which are architecture-specific. However, compared to the relocations available in static linking, there's usually a very reduced number of them.

The other option consists in working exclusively on a single binary and leave all the calls to dynamic functions and memory operations towards dynamic global variables as calls/accesses to dynamic functions/global variables. This would mean that, when, in the translated program, a call to a dynamic function is performed, the control is transferred to the corresponding function in a dynamic library, using the usual dynamic symbol resolution process. This is particularly interesting (and feasible) in case the source and target architectures match (e.g., the input program was compiled for x86-64 and the target architecture is x86-64 too). In case of different target and destination architectures, if the ABI and the prototypes of the library functions are known, it is in theory possible to use a library compiled for a different architecture. However, this could be very dangerous, since a function with the same name, in the same library, could have subtly different semantics when compiled for a different architecture.

The latter option, i.e., employing native libraries, allows us to drastically reduce the performance overhead and the risk of mistranslation on the program as a whole.

However, supporting the usage of native libraries poses a series of challenges, in particular while trying to achieve one of our key goals: independence from the input architecture. In fact, different architectures implement dynamic loading (and lazy loading in particular) in wildly different ways, which we intend to handle in a unified way.

The main challenges are listed in the following.

1. Creation of external symbols. The first step consists in the creation, for each symbol, be it a global variable or a function, of a corresponding entity in our LLVM module. For global variables this means the creation of a global variable with the same name and size, while for functions we simply create a function *declaration* with the same name returning void and taking no arguments. In both cases they have to be marked as having external linkage.

2. Replacement of loads from relocated addresses. We now inspect all the load instructions in the program looking for those accessing an absolute address to which a relocation is associated. Then, we remove the load itself and replace it with the name of the LLVM global variable or function associated with the relocation.

This is coherent with the original behavior of the program, since the name of a function or a global variable in LLVM is used to represent its address.

3. Dispatcher extension. At this point it's possible that the address of one of said functions end up in the global variable representing the program counter. Therefore, the dispatcher (Section 2.2.3) has to be able to handle it properly. For this reason, we expand it to also check if the `pc` contains the address of a dynamic function, and in such case, we emit a direct jump to its address, using inline assembly.

Obviously, the emission of a jump is not enough. In fact, we're switching from a world where the state of the original program is in LLVM global variables, the *translate world*, to a world where the state is expected to be in actual physical registers, the *native world*. Therefore, before the jump, we inject a *serialization* step which uses inline assembly to move the content of each CSV to the corresponding physical register.

This works for transferring the control of the program out to the *native world*, but what about the other way round? For these situation, we have a fallback solution which handles both returns from dynamic functions and also the trickier case of a native dynamic library calling a function pointer belonging to the *translated world*. While the catch-all solution works, it has a cost, therefore we will introduce two other approaches to handle the common case of simple (direct and indirect) calls to dynamic functions.

The catch-all solution. For various reasons, such as function pointers and returns from dynamic functions, the native world could try to jump to the translated world. Since, by design, we leave all the original code where it was, this would mean jumping to the original (non-translated) code, which is unwanted. The only 100% reliable way to handle jumps from the native world to the translated world consists in mapping the memory pages containing the original code as readable, but not executable. In this way, when the native world will try to jump there, it will get a SIGSEGV signal that we can catch. In the signal handler routine, we can perform the *deserialization* step, i.e., copying the content of all the physical registers to the corresponding CSVs, and then simply jump to the dispatcher, therefore resuming execution in the translated world.

Handling of direct calls. To avoid the cost of going through the signal handler and the semantic poorness associated to the previous solution, we optimize the handling of the common case of a typical call to dynamic functions. As shown in Listing 2.3, to do so, we inspect all the uses of the external functions we created and we check if their address ends up in the program counter. If this is the case, we identify the basic block containing the store to the program counter, compute the set of basic blocks it dominates and perform a depth-first search among them looking for function calls. In case we find a function call, we replace it with a proper LLVM call to the corresponding function and a branch to its return address. The LLVM call is then surrounded by the CPU state serialization and deserialization routines.

Handling of indirect calls. A similar approach is followed in case of indirect calls. Every time we identify an indirect function call, instead of branching to the dispatcher directly, we jump to an alternative dispatcher, the *function dispatcher*.

The *function dispatcher* checks if the program counter matches the address of a dynamic functions and, if so, saves the return address in a temporary variable (popping it from the stack, if necessary), calls the CPU state serialization routine, performs an LLVM call to the appropriate dynamic function and, upon return, the deserialization routine is invoked and a branch to the saved return address is emitted.

Once all of these operations have been performed, we only have to make sure that the recompiled program will be linked against the appropriate libraries. Note that, at the current stage, we are exclusively exploring the handling of dynamic libraries in a context where both the input and output architectures match. In fact, using a native version of a library with a program originally compiled for another architecture might lead to serious issues, even leaving ABI-incompatibility aside.

2.3.3 Instrumentation

An obvious application of being able to recompile a program, is recompiling a modified version of it. In fact instrumentation is a key use case of `rev.ng`. In the following we will first describe the LLVM bindings generator for Python and then we will employ it to implement a simple script that instruments all the syscalls of a given program.

`llvmcpy`. To make manipulation of the IR easier, we implemented a binding for LLVM in Python: `llvmcpy` [47]. While implementations of Python bindings for LLVM were already available [6, 75], they were tied to a specific LLVM version, were limited in functionality and some of them kept a Python representation of the LLVM IR, which lead to serious efficiency issues.

For this reason, in `llvmcpy`, we decided to generate bindings for the LLVM framework automatically, starting from the LLVM-C API. The LLVM-C API is a binding for LLVM to be used from C. They feature consistent naming practices, a simple API and, most importantly, they're very stable.

Therefore, by parsing the LLVM-C header files, we can automatically generate an object-oriented view of an LLVM module, without having to represent it in Python. Moreover, due to the consistency of the API and ability to automatically generate the wrappers, `llvmcpy` virtually supports all the past and current LLVM versions out of the box.

Trace syscall numbers example. We will now implement a tracer for syscalls by modifying the LLVM IR recovered by `rev.ng`.

We write a simple Python script which loads the LLVM module, identifies all the syscalls and instruments them injecting the code to print the number of syscall to be performed.

In QEMU for ARM, syscalls are expressed as calls to functions whose name begins with `helper_exception_with_syndrome`. Once we identified all the calls to the said function, we will load the value of the `r7` register, which holds the number of the syscall, and print it to `stderr` using the `dprintf` function.

First of all we need to import `llvmcpy`, obtain the LLVM context object and load the input LLVM IR:

```python
from llvmcpy import llvm
```

```
context = llvm.get_global_context()
path = sys.argv[1]
buffer = llvm.create_memory_buffer_with_contents_of_file(path)
module = context.parse_ir(buffer)
```

Now that we have a reference to the module we can collect the objects required to perform the dprintf call, i.e., the function itself, the CSV representing the register r7, a constant integer representing stderr and the format string for dprintf:

```
r7 = module.get_named_global("r7")
dprintf = module.get_named_function("dprintf")
two = context.int32_type().const_int(2, True)

message_str = context.const_string("%d\n", 4, True)
message = module.add_global(message_str.type_of(), "message")
message.set_initializer(message_str)
int8_ptr = context.int8_type().pointer(0)
message_ptr = message.const_bit_cast(int8_ptr)
```

Note that to build the format string we first have to create a new global variable, then set its initializer to the constant string and finally cast it to char * so that can be passed to dprintf, whose prototype is:

```
int dprintf(int fd, const char *format, ...);
```

At this point we have to iterate over all the instructions of the root function:

```
root_function = module.get_named_function("root")
for basic_block in root_function.iter_basic_blocks():
    for instruction in basic_block.iter_instructions():
        # ...
```

However, we are not interested in all instructions, but only in calls to functions starting with helper_exception_with_syndrome. Therefore, we check the opcode of the instruction, and, if it's a call, we consider the last operand (which represents the called function) and check it's name:

```
if instruction.instruction_opcode == llvm.Call:

    last_operand_index = instruction.get_num_operands() - 1
    callee = instruction.get_operand(last_operand_index)

    if callee.name.startswith("helper_exception_with_syndrome_"):
        # ...
```

Finally, we've found the location where we want to insert our instrumentation. To do this, we create a builder object, position it right before the call instruction, emit an instruction loading r7, prepare the other arguments and, finally, emit the call to dprintf:

```
builder = context.create_builder()
builder.position_builder_before(instruction)
load_r7 = builder.build_load(r7, "")
builder.build_call(dprintf, [two, message_ptr, load_r7], "")
```

The last thing left to do is to serialize the new IR to file:

```
module.print_module_to_file(sys.argv[2])
```

Let's now run our script and recompile the code:

```
$ ./hello.translated
45
45
983045
5
3
6
54
54
4
Hello world!
248
```

We can compare the result with a QEMU run of the original program:

```
$ qemu-arm -strace hello
7346 brk(NULL) = 0x00039000
7346 brk(0x000394b0) = 0x000394b0
7346 open("/dev/urandom",O_RDONLY) = 3
7346 read(3,0xf6ffde84,4) = 4
7346 close(3) = 0
7346 ioctl(0,21505,-151003688,0,221184,0) = 0
7346 ioctl(1,21505,-151003688,1,221184,0) = 0
7346 write(1,0x372a8,13)Hello world!
 = 13
7346 exit_group(0)
```

2.3.4 An Alternative Front-end: CGEN

While QEMU can handle a very large set of architectures, it doesn't support all of them. To further extend the set of architecture handled by rev.ng, it's possible to integrate different front-ends. In this sense, another project which features support for a large set of architecture and accurately describes how each instruction is encoded and its full semantic, is CGEN [55].

CGEN is a project, written in Scheme, part of the GCC umbrella. CGEN allows a developer to describe in a Scheme-inspired language an ISA, with all of its variants, registers, instructions and so on. The encoding of each instruction is also formally specified, along with its behavior. From this high-level description, CGEN can then generate several tools such as disassemblers and emulators.

The idea is to extend CGEN to generate the three key pieces of code necessary for rev.ng to correctly handle an architecture: 1) the code to generate the CSVs, i.e., all the registers and other components of the CPU state, 2) the code necessary to parse raw bytes into an instruction and its fields and 3) the code to generate LLVM IR from a certain instruction. Listing 2.4 presents an example output code for 1) and 2).

CGEN currently supports Morpho Technologies MT Arch, Hitachi SH, Fujitsu FRV, Synopsis ARC, CRIS, Fujitsu FR30, Ubicom IP2K, Lattice Mico32, Renesas M32C, Renesas M32R, OpenRISC 1000, Infineon XC16X, IQ2000/IQ10, Toshiba MeP, Adapteva EPIPHANY and xstormy16.

2.3.5 Extending Platform Support

`rev.ng` currently supports x86, x86-64, MIPS and ARM binaries, statically linked, for Linux. Support for new platforms can be extended in several directions.

First of all, we can extend the number of architectures we handle. Considering the diversity of the ISAs we currently support (in terms of register size, endianess, CISC/RISC design and so on), adding new architectures supported by QEMU is straightforward. As an example, it took a week of work for an inexperienced programmer to add support for x86. As mentioned in Section 2.3.4, additional processors can be supported through alternative front-ends such as CGEN.

In terms of binary formats, `rev.ng` currently handles ELF and has experimental support for PE/COFF. Given its similarity with ELF, support for Mach-O should be straightforward.

The real challenges in terms of fully supporting additional platforms comes from operating systems. Introducing support for FreeBSD is feasible, since QEMU also features a, currently broken, *FreeBSD subsystem*, very similar to the Linux one. Support for Windows requires a much more demanding endeavor which might involve Wine [125].

Supporting execution of bare metal or kernel code is currently outside the scope of `rev.ng`. Note however that, for analysis purposes, `rev.ng` can be a critical tool even on environments not currently supported, such as those just mentioned.

2.4 Performance

After the whole discussion on features and possible use cases, it is interesting to investigate the performance figures of the translated code, in particular when compared with the performances of the original code and of alternative instrumentation frameworks.

Specifically, we decided to compare the performance of the code produced by `revamb` with the original code, with Pin 3.2 [79, 94] (a binary instrumentation framework by Intel for x86 only), QEMU 2.4.50 [14, 97] (Section 1.4) and Valgrind 3.21.0 [46] (an instrumentation framework for building dynamic analysis tools and memory checkers in particular). In the future we also plan to compare our results with other dynamic binary translators such as DynamoRIO [19] and libtdetox [119].

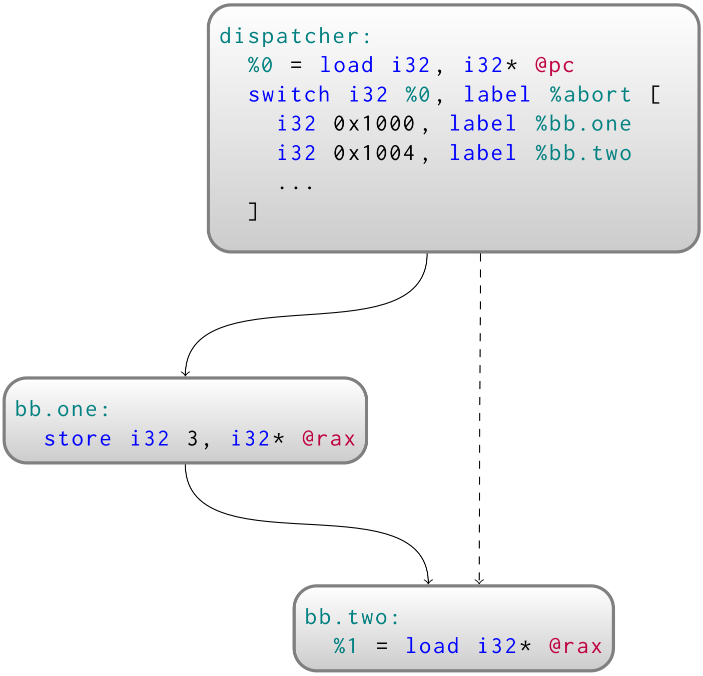
The programs have been compiled for x86-64 using GCC, with `-O2` optimization. They originate from the SPEC2006 [71] integer benchmark suite. The results are obtained over three runs with results computed using the geometric mean [60], running on a server featuring an AMD Opteron Processor 8435 CPU (2.6 GHz, 4 sockets, 6 cores each, 6 MiB of cache), 12 GiB of RAM (DDR2 at 667 MHz) and two WD Green WD20EZRX in RAID-1 mode.

Moreover, we run two sets of benchmarks: a first set with no instrumentation and a second one implementing in all of the tools the *stack distance* instrumentation. The *stack distance* instrumentation is a simple instrumentation which monitors the evolution of the stack pointer (the `rsp` register) and accumulates every change in its value (independently of its sign) in a variable. The final result is the total distance traveled by the stack pointer along the program execution. Comparing the performance of uninstrumented and instrumented code is vital, since certain tools, Pin in particular, in absence of instrumentation, might be able to reuse original code, showing a very low overhead.

Table 2.1 reports the results. The `revamb` produced code is 2.4 times faster than QEMU, almost as fast as Valgrind and 2.8 times slower than native code, while QEMU is 6.6 times slower than native code. For the above mentioned reason, i.e., the ability to reuse strings of original instructions as is, Pin outperforms all of the tools in the uninstrumented cases, with just an overhead around 30% over native code. Specifically, Pin is 2.2 times faster than the code generated by `revamb` in the non-instrumented case.

In case the *stack distance* instrumentation is in place, Pin becomes 17% slower than the `revamb` code. The other relative performance figures remain similar.

Considerations on performance. The speedup of the code generated by `revamb` compared to QEMU (and other tools) is mainly due to the fact that other tools dynamically translate a single basic block at a time. Therefore they have an additional cost to perform the translation at run-time and have limited visibility on the code and, consequently, lesser optimization opportunities compared to our case. In fact, the LLVM optimizer has full visibility on the code and can do a lot more. However, the amount of optimizations that LLVM can perform also heavily depends on the quality of the recovered control-flow graph. In fact, if the CFG is complicated and highly connected, it is, e.g., impossible for an optimizer to perform store-load elision (Figure 2.3). This is currently a problem, due to the fact that the dispatcher can lead execution to any of the translated basic blocks. The CFG can be pruned, but this introduces the risk of breaking certain edges that could be taken at run-time. In the future we aim to evaluate how edges can be pruned from the control-flow graph (in particular edges starting from the dispatcher) preserving correctness. Furthermore, we also want to offer LLVM more optimization chances by detecting functions (Chapter 4) and isolating them in actual LLVM functions (instead of keeping them in the `root` function, see Section 2.2.3). We foresee solid speedups due to this change. We also have a plan to handle misidentification of function boundaries. However, this is outside the scope of this work.



```
dispatcher:
  %0 = load i32, i32* @pc
  switch i32 %0, label %abort [
    i32 0x1000, label %bb.one
    i32 0x1004, label %bb.two
    ...
  ]
```

```
bb.one:
  store i32 3, i32* @rax
```

```
bb.two:
  %1 = load i32* @rax
```

Figure 2.3: Example situation where a spurious edge on the CFG hinders an optimization. Depending on whether the edge from `bb.dispatcher` to `bb.two` is present or not, the optimizer will be able to perform store-load elision in the `bb.one` and `bb.two` basic blocks. Such a transformation would promote `@rax` to an SSA value, which, in turn, will likely be assigned to a register and, therefore, two memory accesses can be saved.

Initial code

```
define void @root(i64) {
bb.main:
  ; 0x4006cf: call 0x4004f0
  ; Code to push return address on the stack
  store i64 4195568, i64* @pc
  call void @function_call(i8* blockaddress(@root, %bb._init.0x8),
                           i8* blockaddress(@root, %bb.main.0x36),
                           i32 4196052)
  br label %bb._init.0x8

bb._init.0x8:
  ; 0x4004f0: jmp QWORD PTR [0x601018]
  %293 = load i64, i64* inttoptr (i64 6295576 to i64*)
  store i64 %293, i64* @pc
  br label %dispatcher.entry
; ...
}
```

Relocations

```
$ readelf -r main
Offset  Type               Symbol's Value  Symbol's Name + Addend
601018  R_X86_64_JUMP_SLOT 0               function + 0
```

Processed code

```
declare void @function()

define void @root(i64) {
bb.main:
  ; 0x4006cf: call 0x4004f0
  ; Code to push return address on the stack
  ; State serialization code goes here
  call void @function()
  ; State de-serialization code goes here
  br label %bb.main.0x36
; ...
}
```

Listing 2.3: Example of handling of a dynamic function call. The first snippet of code provides an example of a dynamic function call in x86-64. `bb.main` calls the stub. The stub then jumps to the address contained in the entry of the `.got.plt` section corresponding to `function` (see Section 1.2.3), which is located at `0x601018`. In the second snippet, we can see that at the corresponding address there's a relocation. `rev.ng` will detect such a situation and produce the code shown in the third snippet. Note how the branch to the PLT stub has been replaced with a function call to `function` (along with the omitted serialization and deserialization code) and, after the call, a branch to the return address has been emitted.

```
class Instruction {
  static unique_ptr<Instruction> make(unsigned int opcode,
                                      uint8_t *code);
  virtual void parse(uint8_t *code) = 0;

  Instruction(unsigned int opcode) : _opcode{opcode} { }

  unsigned int _opcode;
};

struct AddSFormat { unsigned int f_r0, f_r1, f_r2; };

class AddiInstruction : public Instruction {
  void parse(uint8_t *code) {
    sfmt.f_r0 = extractLSB0<unsigned int> (insn, 32, 25, 5);
    sfmt.f_r1 = extractLSB0<unsigned int> (insn, 32, 20, 5);
    sfmt.f_imm = extractLSB0<int> (insn, 32, 15, 16);
  }

  AddiSFormat sfmt;
};

unique_ptr<Instruction> Instruction::make(unsigned int opcode,
                                          uint8_t *code) {
  auto *insn = make_unique<Instruction> { };
  switch (opcode) {
    /* ... */
    case 13 :
      if ((code & 0xfc000000) == 0x34000000) {
        insn = make_unique<AddiInstruction> { opcode };
      } /* ... */
      break;
    /* ... */
  }

  insn->parse(code);
  return insn;
}
```

Listing 2.4: C++ code generated by CGEN to decode and parse the fields of the `add` instruction of the Lattice Mico32 CPU. The `Instruction` class is the base for all the possible instructions. Actual implementations need to implement the `parse` method, which, given a pointer to a memory buffer, populates the fields that compose the current instruction. `AddSFormat` is a structure representing the fields a three-operand instruction. `AddiInstruction` is the actual implementation of the add instruction. It inherits from `Instruction` and embeds `AddSFormat`. Its `parse` method, simply inspects the buffer, selecting the appropriate bits and storing them in the corresponding fields. Finally, the `Instruction::make` method identifies the opcode of the instruction and creates an instance of the appropriate class.

	than native	than Pin	than QEMU	than Valgrind	than `rev.ng`
native is		1.305x faster	6.564x faster	2.743x faster	2.779x faster
Pin is	1.305x slower		5.029x faster	2.101x faster	2.129x faster
QEMU is	6.564x slower	5.029x slower		2.393x slower	2.362x slower
Valgrind is	2.743x slower	2.101x slower	2.393x faster		1.013x faster
`rev.ng` is	2.779x slower	2.129x slower	2.362x faster	1.013x slower	

(a) No instrumentation

	than native	than Pin	than QEMU	than Valgrind	than `rev.ng`
native is		4.098x faster	7.230x faster	3.013x faster	3.491x faster
Pin is	4.098x slower		1.764x faster	1.360x slower	1.174x slower
QEMU is	7.230x slower	1.764x slower		2.400x slower	2.071x slower
Valgrind is	3.013x slower	1.360x faster	2.400x faster		1.159x faster
`rev.ng` is	3.491x slower	1.174x faster	2.071x faster	1.159x slower	

(b) With *stack distance* instrumentation

Table 2.1: Comparison of the performance obtained by Pin [79, 94], QEMU [14, 97], Valgrind [46] and `rev.ng` on the SPEC2006 [71] integer benchmark suite over three runs with results computed using the geometric mean [60]. Table 2.1a shows the comparison running the programs with no instrumentation, while Table 2.1b shows the comparison with the *stack distance* instrumentation.

Chapter 3

Basic Block Identification

As mentioned in Section 2.2.2, accurate enumeration of all the basic blocks in a program is not only critical for an effective translation of the binary, but for analysis purposes too. In this chapter, we propose a systematic approach to identify basic blocks (jump targets) in binary programs by analyzing both the global data and the code itself. The approach is general, as it does not employ heuristics or make architecture-specific assumptions, and is proven effective on a set of real world programs. We also propose a new data-flow analysis (OSRA) particularly suitable to identify jump targets introduced by the sophisticated implementations of `switch` statements. The abovementioned techniques have been implemented in `revamb` and their effectiveness has been assessed on a set of real-world programs on three different architectures with almost no failures due to missing jump targets.

This chapter is in large parts extracted from [49].

3.1 Problem Statement

In this section we present the main challenges in identifying code from a static binary translator perspective, with a specific focus on jump target recovery. In particular, we illustrate a set of problematic cases with examples and investigate their origins.

3.1.1 Identifying Code and Basic Blocks

One of the key issues in static analysis of binary programs consists in isolating the executable code from the program data. Most binary formats contain useful information to this end. As seen in Section 1.2.2, the ELF binary format [131] divides the program in several *segments* that associate a portion of the file to a load address, a size and a set of permissions (such as readable, executable and writable). The permissions are particularly useful, since executable code must reside in a segment with execution permissions.

ELF *sections* would provide a more fine grained information, but since, unlike *segments* they're not critical to execution, they are often absent. This becomes an

```
cmp      r0, #240
addls    pc, pc, r0, lsl #2
b        21304
b        21320
b        21710
b        212fc
```

(a) $\text{pc} + 4 \cdot r0$, with $r0 < 240$

```
cmp      r3, #8
ldrls    pc, [pc, r3, lsl #2]
b        128c4
.word    0x1265c
.word    0x124ec
.word    0x12518
```

(b) $mem[\text{pc} + 4 \cdot r3]$, with $r3 < 8$

```
cmp cl,0x53
ja  471aa8
lea rax,[rip+0x3c9ca]
mov rcx, PTR [rax+rcx*4]
add rax,rcx
jmp rax
```

(c) $\text{base} + mem[\text{base} + 4 \cdot cl]$, with $cl \leq 83$

Listing 3.1: Two ARM and an x86-64 real-world implementations of the `switch` statement. Listing 3.1a presents an implementation where the new address is written directly in the program counter (`pc`) and is computed as the current PC plus the switch value `r0` left shifted by two positions. The code in Listing 3.1b reads the jump target from an array of addresses (a *jump table*) stored in a constant pool close to the current program counter (the `.word` directives). The chosen address is determined using the switch value `r3` as an index. In Listing 3.1c we have an x86-64 `switch` implementation reading a value from $\text{base}+4{\cdot}cl$ (where $\text{base} = \text{pc}+$`0x3c9ca` and cl is rcx's lowest byte), which is then added to base and used to perform an indirect jump. Note how, in all the examples, before computing the target address, the switch value is compared with a constant.

issue when, as it is often the case, the linker merges `.rodata` and `.text` in a single segment, since they both have read-only access.

Furthermore, code and data can be mixed by the compiler, e.g., when *constant pools* are used in unified cache architectures to reduce the cost of loading constants. Once the problem of distinguishing code from data is solved, basic blocks must be identified to reconstruct the control flow. Basic blocks are delimited by instructions that alter the control flow (branches, jumps and calls), or by *jump targets* – corresponding to labels in the assembly code. It is worth noting that in static binary translation control flow reconstruction could theoretically be avoided. However, there are significant drawbacks if this choice is taken. In particular, control flow reconstruction enables more aggressive optimizations. In the absence of this information, every instruction must be considered as a basic block on its own.

Moreover, in architectures employing a variable-length instruction encoding (VLE), such as x86, a single sequence of bytes would have to be interpreted in several different ways, leading to a needless increase in translation time and output size.

Finally, a relevant concern is also the way indirect jumps are typically handled in static binary translators [139]. If it is not possible to statically identify the target of an indirect jump, a static binary translator will defer that decision at run-time, by diverting the execution to some form of dispatcher that will jump to the code generated corresponding the correct instruction. The dispatcher can be implemented through a hash table or a binary search tree, but in all cases, the more the possible jump targets, the more space and/or time to handle an indirect jump is required.

In conclusion, identifying the possible jump targets is a key problem, and while a small amount of false positives is acceptable, marking all the instructions as a possible jump targets can seriously affect the performances of the translated binary.

3.1.2 Challenges in Jump Target Recovery

The target of a jump instruction can be either encoded directly in the jump instruction (a direct jump) or can be the content of a register or memory area at run-time (an indirect jump). Direct jumps can be either relative to the program counter (PC) or absolute, in which case the immediate represents the full destination address. In both cases, obtaining the jump target is straightforward. On the other hand, indirect jumps deserve a closer analysis, as they can derive from several different types of high level statements. In the following we present the most relevant ones.

Materialized destination address. A program might need to perform a jump relative to the PC to an address whose distance from the PC is larger than the maximum representable in the instruction immediate. An option to circumvent this situation is presented in the following MIPS snippet:

```
lui     t9,0x42
addiu   t9,t9,0xd188
jr      t9
```

In the example the full destination address is materialized in a register, and then an indirect jump through that register is performed. Another possible solution consists in storing the full target address in a constant pool.

Return instructions. A return instruction is a form of indirect jump that diverts execution to the address stored in the link register or on the top of the stack.

Function pointers. Calling a function pointer or a C++ virtual function also requires an indirect jump.

`switch` statement. Dense `switch` statements are typically implemented using indirect jumps through a register which typically contains an address dependent on the switch value. Listing 3.1 reports the code emitted due to a switch statement in three different real-world cases.

From the point of view of a static binary translator, the most challenging indirect jumps are those produced by `switch` statements. In fact, their destination address is often computed at run-time and therefore it’s never explicitly available in the code or data segments (see Listing 3.1a and Listing 3.1c).

3.2 Harvesting Data and Code

In this section we will introduce three methods for the harvesting of jump targets. The first will inspect the global data, while the remaining two work on the code directly.

3.2.1 Global Data Harvesting

The program global data can provide useful information for jump target recovery. In fact, the `.rodata` and `.data` sections often contain function pointers, C++ virtual tables, or jump tables (see Listing 3.1b). Constant pools (see Section 3.1.2) can also be a source of pointers to basic blocks or functions, in case of jump instructions targeting addresses not reachable via an immediate offset added to the PC.

For this reason, the most straightforward approach to recover an initial set of jump targets consists in traversing byte-by-byte (or word-by-word) all the program segments looking for *code pointers*.

A *code pointer* is a byte sequence of the length of a pointer (32 or 64 bits) that, when interpreted using the appropriate endianess for the architecture, represents an address lying within an executable segment. If the ISA enforces an instruction alignment, e.g., 4 bytes in ARM, the resulting address must also be aligned to that value.

3.2.2 Simple Expression Tracker

Once an initial set of jump targets is available, the code at the corresponding addresses is translated. We therefore need to introduce an analysis aimed at harvesting jump targets from the translated code. The analysis we propose, called Simple Expression Tracker (SET), identifies all the store instructions and tracks in a step-by-step way how the value being stored is computed. The analysis proceeds as long as the operations composing the expression depend at most on a single non-constant operand. In fact, SET aims to collect the destination address of direct jumps and indirect jumps that materialize the destination address in multiple instructions. Consider the example of such an indirect jump in Section 3.1.2: SET will detect that the indirect jump (represented as a store to the CSV `t9`) targets `0x42d188`.

Algorithm 3.1 describes SET. When the SET processes a `store` instruction, it creates an empty stack, the *operations stack*, it inspects the value to be stored and proceeds differently depending on its type.

If the value being stored is the result of a binary operation i (e.g., a subtraction) and its second operand j is constant, we record on the stack the $\langle i, j \rangle$ pair. Then, we proceed considering the non-constant operand.

If the stored value is the result of a load operation from an unknown memory location (i.e., not from a CSV), we record the load operation on the stack and proceed to analyze how the address of the load operation is computed.

The analysis repeats the same operation with the newly considered operand, progressively growing the stack by pushing new operations. The process terminates when an instruction that cannot be handled is met (e.g., an addition with no constant operands) or when the operation to consider is a constant k, which means that a load from a constant address has been found or that both operands of the binary operation are constant.

In the latter case, it's possible to materialize a constant value: a variable n is initialized with k and the *operations stack* is traversed from top to bottom, updating the value of n by combining it with the operation registered on the stack. Therefore, for a load operation, if n is an address contained in a segment of the binary, its value is updated with the content of the pointed memory area, when this is statically available. For a binary operation, the new value will be the result of performing the operation i using n and j as operands.

Load/store handling. In addition to this, the SET explicitly handles loads from CSVs. A load from a CSV can be affected by multiple stores, therefore, to process them we employ a LIFO worklist. When a load from a certain CSV is met, the analysis proceed backwards, starting from the load instruction, looking for store instructions writing to that CSV and exploring recursively all the ancestor basic blocks until such a store instruction is found. For each found store instruction, a pair $\langle s, h \rangle$ is pushed on top of the worklist, where s is a reference to the store instruction, and h is an integer number representing the current height of the *operations stack*. The analysis proceeds by processing the element on the top of the worklist. When the top element is extracted from the worklist, the *operations stack* is cut to height h and the analysis proceeds from the value stored by s. This is necessary to restore the stack to its state when the work item was inserted into the worklist, discarding all the operations pushed on the stack while processing other work items.

The advantage of this approach lies in the fact that, by using a depth-first exploration, we can always reuse the lower part of the *operations stack*, without ever duplicating it, with the net effect of keeping its size in $O(n)$ of the number of instructions.

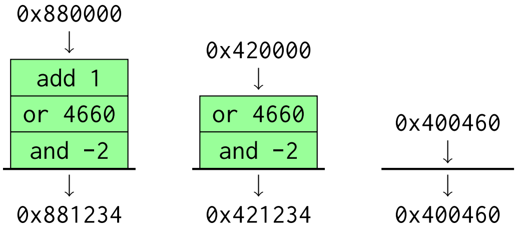
This analysis is very effective in collecting the simplest jump targets hidden in the code. More specifically, it can collect the destinations of direct jumps, indirect jumps with a constant destination materialized in a register (see Section 3.1.1) and also all the return addresses of call instructions. In fact, as mentioned in Section 2.2.5, the generated code doesn't have the concept of *call instruction*, but represents them as a simple write to the program counter preceded by an instruction storing the return address on the stack or in the link register. In both cases, since we track all the stores and not only those to the PC, our analysis is able to catch the return address. Figure 3.1 presents some examples of jump targets recovered from the code through the SET.

3.3 The OSR Analysis

Despite its effectiveness, SET is not able to collect jump targets due to switch statements such as those shown in Listing 3.1. In fact, in these cases, the jump target depends on a non-constant operand: the result of the expression evaluated by the `switch` statement. Therefore, we introduce a specialized data-flow analysis

```
    lui   v0, 0x42
    ble   a0, t0, callee
    nop
    lui   v0, 0x88
    addi  v0, 1
callee:
    ori   v0, 0x1234
    jal   v0
```

(a) Input MIPS assembly



(b) Schematization of the SET jump target recovery

```
  store i32 0x420000, i32* @v0
  %1 = load i32, i32* @t0
  %2 = load i32, i32* @a0
  %3 = icmp slt i32 %1, %2
  br i1 %3, label %call, label %fallthrough

fallthrough:
  store i32 0x880000, i32* @v0
  %4 = load i32, i32* @v0
 |%5 = add i32 %4, 1
  store i32 %5, @v0
  br label %call

call:
  %6 = load i32, i32* @v0
 |%7 = or i32 %6, 0x1234
  store i32 0x400460, i32* @ra
 |%8 = and i32 %7, -2
  store i32 %8, i32* @pc
  br label %dispatcher
```

(c) LLVM IR produced by the `revamb`

Figure 3.1: Example of the SET algorithm. Figure 3.1a shows a MIPS assembly snippet of an indirect function call with two possible targets (`0x881234` and `0x441234`). In the example, three jump targets can be recovered: the return address being stored in the link register `ra` by the function call (`jal`) and the two possible destinations of the function call, stored in the `v0` register. In Figure 3.1c the LLVM IR produced by `revamb` is presented along with the two paths leading to the creation of a jump target: both start from a store to the program counter CSV, then, they split in the load `%6` and end in two distinct store of constant values. SET traverses these two paths and records in the *operations stack* all the instructions it meets, except for CSV-related load/stores (notice the vertical bars on the left of the recorded instructions), until an instruction where all the input operands are constant is met, i.e., the constant store instructions. At this point the *operation stack* is traversed from top to bottom executing the registered operations to compute the jump target. Figure 3.1b shows the state of the *operations stack* when the constant is met at the end of two paths, along with a zero-height stack due to the constant store in the link register CSV `@ra`.

```
Data: A store instruction $s$
Result: A jump targets generator
$Ops = ()$;
$Worklist\,(\langle s, 0\rangle)$;
while $Worklist$ is not empty do
    $c$, $h$ = pop($Worklist$);
    truncate $Ops$ to $h$ elements;
    $next$ = getStoredValue($c$);
    while $next$ is set do
        $i = next$;
        unset $next$;
        if $i$ is a binary operation $v \circ j$ then
            if $j$ is constant and $v$ is not then
                push $\langle i, j\rangle$ onto $Ops$;
                $next = v$;
        else if $i$ is a load instruction from $a$ then
            if $isCSV(a)$ then
                foreach $w$, previous store to $a$ do
                    push $\langle w$, getHeight($Ops$)$\rangle$ onto $Worklist$;
            else
                push $i$ onto $Ops$;
                $next = a$;
        else if $i$ is a constant value then
            $n$ = value($i$);
            foreach $\langle o, j\rangle$ in Ops, top to bottom do
                if $o$ is binary operation then
                    $n$ = apply($o$, $n$, $j$);
                else if $o$ is a load instruction then
                    $n$ = readFromSegment($n$);
            yield $n$;
```

Algorithm 3.1: The Simple Expression Tracker algorithm.

whose goal is to try and represent each SSA value in the following form:

$$a + b \cdot x, \text{ with } \left\{ x : \begin{array}{c} c \leq x \leq d \\ x < c,\ x > d \end{array} \text{ and } x \text{ is } \begin{array}{c} \text{signed} \\ \text{unsigned} \end{array} \right\}$$

where: a is a constant base value, b is a constant scaling factor and x is a reference to a *free* SSA value associated with a (possibly negated) range $[c, d]$ and a signedness (signed or unsigned).

We chose this form as the optimal trade-off between complexity and expressive power to model how the destination address of a `switch` statement's indirect jump is computed. In particular, it's suitable to capture the jump targets represented in Listing 3.1 or part of it (e.g., pc + $4 \cdot i$ with $i < 8$). Any increase in terms of expressive power would raise sensibly the complexity of the analysis and, as we will see, it wouldn't produce any benefit.

We define x, together with its constraints, as a *bounded value* (BV). We also an instance of the above expression an *offset shifted range* (OSR). We therefore call our analysis *OSR analysis* (OSRA).

A BV is always associated with an SSA value x which cannot be expressed in terms of an OSR relative to any other SSA value. In other terms, a BV represents a *free* SSA value associated to a range constraint. An example of such free SSA value might be the result of a *xor* operation, which exceeds the expressiveness of an OSR. The OSRA traverses all the program instructions and, where possible, associates them with an OSR.

In parallel, the analysis also tracks constraints that hold in a certain basic block in the form of BVs. To this end, the analysis processes comparison instructions and tracks their usage in conditional branch instructions. For example, if an instruction performs an unsigned comparison to check if an SSA value x is less than or equal to 7, OSRA will create a BV $\{x : 0 \leq x \leq 7, \text{unsigned}\}$ and will associate it with the comparison instruction. If the result of the comparison is then used in branch conditional instruction, the analysis will associate the BV with the basic block taken if the condition holds, and all the OSRs relative to x in this basic block will be affected.

3.3.1 OSR Tracking

Initially, no instruction is associated with an OSR. When OSRA is given an instruction i representing a binary operation, the number of non-constant operands is checked, and, as with SET, if more than one is present, it is ignored. Otherwise, the non-constant operand is considered. If it has an OSR, it is used as a base for the new OSR. If it doesn't have an OSR, a *basic OSR* is created. A *basic OSR* is an OSR with $a = 0$, $b = 1$ and x is set to a BV representing the non-constant operand. In both cases, the resulting OSR has to be updated according to the semantics of the current instruction (Table 3.1), and the constraints on the BV are updated for the new context. In fact, if the non-constant operand and the current instruction are in two distinct basic blocks, the constraints on a BV might be different. If i's basic block is not already associated with a BV for x, a new one is created without constraints (i.e., it is set to $\top$), otherwise, the existing one is used. Note that,

Op	Resulting OSR	Op	Resulting OSR
$+$	$(a + k\ +x \cdot b$	$/$	$(a/k\ +x \cdot (b/k)$
$-$	$(a - k)+x \cdot b$	$\ll$	$\left(a \cdot 2^k\right)+x \cdot \left(b \cdot 2^k\right)$
$\times$	$(a \cdot k\ +x \cdot (b \cdot k)$	$\gg$	$\left(a/2^k\ +x \cdot \left(b/2^k\right)$

Table 3.1: Effect of composing an OSR $a + b \cdot x$ with a constant k through a binary operator.

unlike the SET, OSRA does not support all the possible instructions, but only the subset that can be handled considering the OSR expressive power. For example, the *rotate* and *xor* instructions cannot be handled.

It is important to understand that by cloning the OSR of the non-constant operands, the associated BV is also being propagated. This means we can have several instructions, possibly one using the result of the other, possibly on different execution paths, all expressed with respect to a single SSA value. This is particularly beneficial, since if OSRA is able to verify that in a certain set of basic blocks an SSA value, or an OSR referring to it, is constrained in some way (e.g., has an upper bound), all the OSRs using it can benefit from this information directly.

3.3.2 BV Tracking

A BV tracks, for a certain SSA value x, the lower and upper bound of the range within which x lies, possibly negated, and its *signedness*. By *signedness* we mean whether the SSA value represents a signed or unsigned integer, the *sign* itself is not tracked. The initial value of a BV is $\top$: it can assume any value and has an *unknown* signedness.

Each basic block is associated with a set of BVs which are known to hold for that basic block. Also certain instructions can be associated with BV, indicating their run-time result represents whether the associated BV (i.e., constraint) holds or not. This information becomes useful when the result of the instruction is used as the condition for a conditional branch instruction. In fact it is possible to state that in the basic block taken if the condition is true the constraint of the BV holds, while in the successor it does not.

To track BVs, OSRA considers three types of instructions: comparisons with constants, logical and/or instructions and conditional branches.

Comparisons and signedness. When a comparison with a constant k is met, the OSR associated with the non-constant operand is considered, or, if it doesn't have one, a *basic OSR* referred to the operand itself is created. The expression represented by the OSR is then compared with the appropriate comparison operator (e.g., signed greater than or equal) with the constant operand, obtaining a first-degree inequation.

$$a + b \cdot x \geq k \Longrightarrow x \geq \frac{k - a}{b}$$

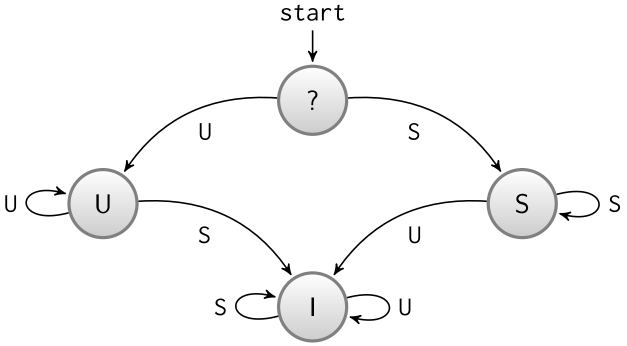


Figure 3.2: Finite-state machine representing the possible signedness state transitions of a BV. The `?`, `U`, `S` and `I` nodes represent respectively an *unknown* signedness, an *unsigned* value, a *signed* value and a value with an *inconsistent* signedness. The edges represent the transition performed when the value associated to BV is used with an unsigned (`U`) or signed (`S`) operation.

The solution is then used as a constraint on x and a new BV is created and assigned to the comparison instruction.

Moreover, if the comparison is not simply a check for equality or inequality, it carries a signedness information, i.e. it can be signed or unsigned. This information is propagated to the BV corresponding to x associated to the basic block, which updates its signedness according to the finite-state machine in Figure 3.2. The signedness of a BV affects the maximum upper bound and the minimum lower bound, which are those of an `unsigned int` for an *unsigned* BV, those of a `signed int` for a *signed* BV and an intersection of the two in case of *inconsistent* signedness.

The signedness is particularly relevant for our purposes, since the lower bound of an unsigned BV is implicitly zero, and therefore with a single additional constraint (e.g., $x \leq 5$) we can limit the value of x in small range, which is desirable to the final aim of the analysis.

Logical operators. The second type of instruction handled by OSRA to track BVs are the logical *and* and *or* operators. If both instruction operands are associated to a BV referring to the same SSA value, the two constraints are merged according to an *and* or *or* policy depending on the instruction. The merge policy considers the constraints as ranges (possibly negated) and combines them through the union operation (*or* merge policy) or the intersection operation (*and* merge policy), and generates a new constraint which is then associated with the instruction. However, the merge operation can fail, for instance if two positive disjoint ranges have to be combined with using the *or* merge policy, since the result exceeds the expressive power of the BV, which can represent at most a single positive range. In this case, the BV is set to $\perp$.

Conditional branches. The most important instruction type for tracking BVs are conditional branches, since they allow the analysis to state that a certain constraint, or its opposite, holds in a certain basic block. More specifically, when a conditional branch instruction is analyzed, if the SSA value used as a condition is associated with a BV, this BV is propagated to the first successor (the *true* branch) and its negated form is propagated to the second one (the *false* branch).

Therefore, each basic block is associated with a set of BVs obtained by propagation from its predecessors. Since a basic block might have multiple predecessors propagating different constraints, the BV considered to hold in a basic block is obtained as the result of a merge operation of the BVs coming from each predecessor using the *or* policy. Moreover, since a single predecessor might propagate a BV multiple times (a basic block might be analyzed more than once), OSRA explicitly registers which BV has been received from which predecessor. This way, the BV coming from a predecessor can be updated and it is possible to recompute without information loss the resulting BV for the basic block by *or*-merging all the BVs again. Note that if a predecessor does not provide a constraint for a certain SSA value, it is assumed to be unconstrained, and therefore the merge operation will produce a $\top$ value. Note also that in case a certain predecessor propagates a BV relative to a certain SSA value multiple times, the new constraint will be at least as strict as the previous one.

3.3.3 Load and Store Handling

To increase the effectiveness of the analysis, we also keep track of load and store instructions targeting CSVs. In particular, we have two objectives. First, we need a form of *reaching definitions analysis* to propagate OSRs and BVs being stored to a CSV to all the load instructions reading that value. For instance, the result of a compare instruction might be saved in a CSV, and therefore the associated constraint needs to be propagated to all the load instruction reached by that store. Second, even if the analysis cannot track what is being loaded, we want to be able to express the fact that two instructions loading the same CSV, among which a path exists without instructions writing to that CSV, are loading the same value.

To this end, when a store or load instruction using the CSV r is met, its OSR gets propagated, or, if it doesn't have an OSR, a new *basic OSR* is created referring to r. Propagation takes place by recursively exploring the subsequent instructions in the basic block and in its successors, looking for load instructions reading r, until a store to r is met. If a BV is associated to r, it is propagated too.

While propagating a load or a store, the analysis keeps track of which load instruction were affected by a the propagation in the *overtaken* set. If, while propagating a load or a store, a load instruction already associated with an OSR is met, a check on the OSR is performed: if its BV is part of the *overtaken* set, the propagation takes place and the existing OSR is overridden, otherwise it means the load instruction depends on multiple BVs. In the latter case, we do not have a merge policy and simply stop the propagation. The existing OSR is replaced with a self-referencing *basic OSR* identified with $\perp$, which will prevent any future propagation.

On the contrary, while propagating a BV, if a load instruction already associated with a BV referring to the same SSA value is found, the two constraints are merged using the *or* policy.

3.3.4 Integration with SET

As discussed, the primary aim of OSRA is to recover jump targets for a certain type of `switch` statements. However, while it provides useful information to this end, compared to the previously presented analysis, it presents some shortcomings: it cannot read data from memory segments present in the binary and can only handle a subset of all the possible binary operations. For this reason, we enhanced the SET to exploit the information provided by OSRA. The integration with OSRA affects two aspects: constant handling and materialization of OSRs.

For the former aspect, while describing the SET, we mentioned that it was able to handle operations with at most a single non-constant operand. Thanks to the OSRA we can expand the concept of *constant* to SSA values associated with an OSR whose BV is constrained to a single value (i.e., the lower and upper bounds match). This opens up for handling a slightly larger amount of situations.

The second, and most relevant, aspect is the OSR materialization. If, while building the *operations stack*, an instruction that cannot be handled is met, the analysis checks if an OSR is available for that instruction. If so, we compute $min = a + b \cdot c$ and $max = a + b \cdot d$. Then, all the operations on the *operations stack* are applied to them. If, in both cases, the result is a valid *code pointer*, then the OSR can be used to produce jump targets. Therefore, all the values that the OSR represents are generated, from *min* to *max* with a step size of b, and go through the operations in the *operations stack*, producing all the jump targets represented by the OSR.

In Figure 3.3 the code generated by an ARM compiler for a `switch` statement is exemplified and annotated with the information produced by OSRA. The example shows most of the features of the analysis we discussed, such as propagation of stored values (`%2`→`%4`) and merge of BVs coming from multiple predecessors (`BB3`). Note that `%8` holds the key information to obtain 5 jump targets, but, since OSRA does not handle logical and on OSRs (`%9`), the SET is necessary to let the information associated to `%8` reach the PC store.

3.4 Experimental Results

All the presented techniques have been implemented in our static binary translator, `revamb`.

For our experiments, we tested three popular input ISAs: MIPS, ARM and x86-64. This choice was guided by the attempt to test the various features an ISA can have, such as: endianess (MIPS is big endian, the others are little endian), register size (32 or 64 bits), CISC vs RISC designs, variable-length instruction encoding (x86-64) and delay slots (MIPS). For ease of testing, we chose as destination architecture x86-64 in all cases. For our tests, the following toolchains have been employed: GCC 5.3.0 using uClibc for ARM and musl for MIPS, and GCC 4.9.2 with musl for x86-64. All the tests applications were linked statically. Note that static binaries provide less information than dynamically linked executables, since the dynamic table and the dynamic symbols are absent. This also means that our tool handles the C standard library, which tends to be very large, include hand-

```
BB1:
  %1 = load i32, i32* @r1
  %2 = sub i32 %1, 4 ; [-4 + 1 * %1]
  store i32 %2, i32* @ZF
  %3 = icmp uge i32 %1, 4 ; (%1, u, 4, max)
  br i1 %3, label %BB2, label %BB3

BB2: ; (%1, u, 4, max) = <BB1, (%1, u, 4, max)>
  %4 = load i32, i32* @ZF ; [-4 + 1 * %1]
  %5 = icmp ne i32 %4, 0 ; (%1, u, 5, max)
  br i1 %5, label %exit, label %BB3

BB3: ; NOT (%1, u, 5, max) =
     ;        <BB2, (%1, u, 4, 4)>
     ;     || <BB1, NOT (%1, u, 4, max)>
  %6 = load i32, i32* @r1 ; [0 + 1 * %1]
  %7 = shl i32 %6, 2 ; [0 + 4 * %1]
  %8 = add i32 113372, %7 ; [113372 + 4 * %1]
  %9 = and i32 %8, -2
  store i32 %9, i32* @pc
```

Figure 3.3: Example of the LLVM IR generated by two ARM instructions: `cmp r1, #5; addls pc, pc, r1, lsl #2`. Comments indicate information produced by OSRA, in particular (x, s, c, d) represents a BV, $[a + b \cdot x]$ an OSR and $(BV) = \langle BB1, BV1\rangle \,||\, \langle BB2, BV2\rangle$ the BV associated to a basic block, obtained by *or*-merging BV1 (coming from BB1) and BV2 (coming from BB2).

written assembly and other sophisticated pieces of code which are not found in ordinary binaries. In summary, using static binaries, puts us in the most difficult setting.

The translation process. `revamb` translates the input binary in an iterative fashion. The translation starts from the entry point of the program and all the jump targets that have been found in the ELF segments, as described in Section 3.2.1. Once a whole basic block has been translated, direct jumps (i.e., constant stores to the program counter CSV) and fall-through jump targets are automatically detected and added to the list of addresses to visit. When the code at all the known addresses has been consumed, the Simple Expression Tracker LLVM pass (see Section 3.2.2) is run and all the harvested jump targets are processed. SET is run repeatedly on the new code, until it doesn't produce any new jump target. At this point, the OSRA LLVM pass (see Section 3.3) is executed over the generated code and the collected jump targets are explored. The process is iterated until no more jump targets can be recovered: the generated LLVM IR is then considered complete and ready for optimization and compilation.

3.4.1 Functional Testing

The first and foremost objective of revamb is to produce working binaries. Therefore, willing to asses the effectiveness of our approach, we took the coreutils project, a set of 104 popular command line utilities such as ls, base64, md5sum and many others, and translated its binaries. Then, we run the 567 tests in the coreutils test suite on the binaries translated by revamb with and without OSRA enabled. The translation process and the tests were run on several different machines with different characteristics. On a Linux-based machine with an AMD Opteron 8378 CPU and 32 GiB DDR3 RAM, the average translation time of an ARM program (305 kiB on average) was approximately 110 seconds.

Due to some limitations in syscall management, we expected some failures, in particular due to the absence of support of multithreading, forking and a couple of other syscalls. However, in this paper our main aim is to identify jump targets correctly. Therefore, the most relevant result is how many tests failed due to an unhandled jump target. Fixing the remaining issues is mostly a matter of engineering work in improving the integration with the QEMU's syscall translation subsystem, and lies outside the scope of this work.

Table 3.2 summarizes the results on the coreutils test suite. Enabling OSRA, the amount of passed test moves from the 51%/57%/85% of the total to 65%/82%/85%, on MIPS/ARM/x86-64 respectively. The difference is more sensible in non-VLE architectures, since switch statements are easily translated in the form presented in Listing 3.1a, while on x86-64 most of them are implemented using jump tables, which are easily caught by the global data harvesting pass described in Section 3.2.1.

Apart from the raw amount of passed tests, the key point to consider to evaluate the effectiveness of the analysis we developed, is the amount of programs part of the test suite failing due to an unidentified jump target. The "U" column in Table 3.2 shows how their number is very low even employing only SET (5 programs in MIPS and 3 in ARM), and reaches to 0 in all cases using OSRA, with the exception of MIPS. The failures in the MIPS case are due to code similar to the following:

```
lui     s3,0x40
bal     412120 ; Delay slot omitted
addiu   s3,s3,0x0a48
```

In this case, the SET wasn't able to catch the value being stored in s3 because it is built in part before a function call and in part afterwards.

3.4.2 Basic Block Size

A naïve approach to identify all the jump targets is to mark all the addresses in the executable segment as jump targets, but, as discussed in Section 3.1.1 this approach has several drawbacks.

Therefore, to assess how our solution performs compared to the naïve approach, we computed the average length of translated basic blocks, or, in other terms, the average distance in instructions among one jump target and the next one.

In Table 3.2 we can see the average length is well above the average length expected in the naïve approach, 1 or even less in case of VLE ISAs. Note that in the computation of the average length of a basic block, we ignored overtranslated portions of the binary, since optimizing code that will never be executed is not useful.

	Coverage						Tests (SET only)				Tests (OSRA)				Jump targets	
	Covered	Unused	NOPs	Other	Extra	IPB	S	P	F	U	S	P	F	U	Total	OSRA
MIPS	95.37%	4.61%	0.00%	0.02%	12.51%	5.17	169	228	170	5	160	266	141	3	1441834	49540
ARM	89.56%	8.91%	0.13%	1.40%	14.16%	3.98	140	220	207	3	143	350	74	0	807323	61190
x86-64	94.84%	4.70%	0.46%	0.00%	12.87%	4.22	163	343	61	0	163	343	61	0	1162082	23731

Table 3.2: Statistics concerning the translation of `coreutils` binaries. The “Coverage” column reports the average percentage of the original code that has been translated (“Covered”) and that has been ignored because belongs to an unused function (“Unused”), it is composed exclusively of no-op instructions (“NOPs”) or for other reasons (“Other”). The “Extra” column represents the amount of *overtranslated* code with respect to the actual code. “IPB” represents the average number of instructions per basic block. The “Tests” columns report the results of running the `coreutils` test suite using binaries produced by `revamb` without and with OSRA enabled. The amount of skipped (S), passed (P) and failed (F) tests is reported for each situation. A test is skipped if the test suite detects that the environment doesn’t meet the conditions to run the test. The “U” column, highlighted in gray, represents the amount of `coreutils` programs failing due to a missing jump target. Finally, the total amount of jump targets collected over the whole test suite is reported, along with the number of those recovered thanks to OSRA.

3.5 Conclusions

We have introduced two methods to recover jump targets from binary code, Simple Expression Tracker and OSR Analysis, respectively targeting the values used in store instructions and the jump targets generated by `switch` constructs in the source code. The proposed analyses have been implemented in `revamb`, a static binary translation framework based on LLVM and QEMU. An experimental campaign on the `coreutils` binaries, compiled for ARM, MIPS and x86 targets, shows that the proposed analyses provide a coverage of the jump targets ranging from 89.56% to 95.37% depending on the target architecture.Furthermore, OSRA proves particularly effective on ARM binaries, increasing the number of tests successfully completed by more than 50%. In the future we plan to tackle more directly the *overtranslation* problem and, most importantly, to integrate in our framework function recognition to further improve our results.

Chapter 4

CFG and Function Boundaries Identification

Static binary analysis is a key tool to assess the security of third-party binaries and legacy programs. Most forms of binary analysis rely on the availability of two key pieces of information: the program's control-flow graph and function boundaries. However, current tools struggle to provide accurate and precise results, in particular when dealing with hand-written assembly functions and non-trivial control-flow transfer instructions, such as tail calls. In addition, most of the existing solutions are ad-hoc, rely on hand-coded heuristics, and are tied to a specific architecture.

In this chapter we first highlight the challenges faced by an architecture agnostic static binary analysis framework to provide accurate information about a program's CFG and function boundaries without employing debugging information or symbols.

Then, we propose a set of fully automated principled analyses, built on top of our framework to recover an accurate CFG and function boundaries, even in presence of hand-written assembly and neither employing ISA-specific (hand-crafted) heuristics nor relying on debugging information or symbols. Our solution handles effectively predicate instructions, `noreturn` functions, tail calls, and context-dependent CFG.

In the evaluation, we test our tool on binaries compiled for MIPS, ARM, and x86-64 using GCC and clang and compare them to the industry's state of the art tool, IDA Pro, and two well-known academic tools, BAP/ByteWeight [20, 11] and angr [142, 141]. In all cases, the quality of the CFG and function boundaries produced by `rev.ng` is comparable to or improves over the alternatives. We also demonstrate that we handle the CFG of non-trivial, real world hand-written assembly functions where other tools fail.

This chapter is in large parts extracted from [51].

4.1 Challenges

Recovering the *control-flow graph* and the function boundaries of a binary program for which no source code or debugging information are available presents several challenges [158]. In the following, we discuss them focusing on the issues that an ISA-independent analysis framework faces.

4.1.1 Challenges in CFG Recovery

Recovering the CFG of a program consists in essentially two phases. The first phase, thoroughly describe in Chapter 3 identifies the basic blocks composing the application, while the second phase establishes the correct relationships among them in terms of control-flow. In practice, this means that the analysis has to enumerate all basic block starting addresses, their size, and whether a control-flow transfer from one basic block to another is feasible or not.

After collecting basic blocks' starting address and size, the control-flow recovery can start. Direct jump instructions provide useful information and are straightforward to incorporate. Unfortunately, they are not sufficient to completely recover the CFG. The main challenge consists in handling indirect control-flow transfer instructions, i.e., indirect function calls and indirect branch instructions. In general, it is impossible to enumerate the exact set of possible *jump targets* for indirect control transfers. The worst case is represented by a jump to a user-controlled value, where all executable code becomes reachable. Alternatively, the destination address may be the result of an arbitrarily complicated computation, which might even be impossible to track statically.

With this premise, we classify *indirect* control-flow transfer instructions in three categories that, when handled correctly, provide an accurate CFG:

Compiler-generated, function-local CFG. All the indirect jump instructions generated by a compiler to efficiently lower the control-flow of certain statements, most notably C `switch` statements.

"Reasonable" hand-written assembly. Indirect jumps manually introduced by the developer in assembly, usually to optimize low level routines such as `memcpy`. Compared to the CFG that a compiler typically generates, the developer can produce more efficient but hard-to-analyze code. By *reasonable*, we mean, for instance, a function that does not make assumptions about the values of its parameters, but rather enforces (implicitly or explicitly) the constraints locally (see example in Section 4.3.2).

Indirect function calls. Indirect function calls through function pointers or C++ virtual functions.

In this work, we aim to handle accurately the first case and to develop a set of analyses to handle as many cases as possible of the second category. In fact, these two classes make up the most part of the CFG needed for our final goal: the recovery of function boundaries. On the other hand, indirect function calls might involve virtual tables or function pointer fields of dynamically allocated objects, which are harder to track statically. In short, recovering targets of function calls is an orthogonal problem, which deserves to be treated on its own and does not compromise functionality, since pointer to targets of indirect function calls are typically available elsewhere (e.g., virtual tables). Note that, to enable support for large binaries, as a design choice, we do not employ SMT-solvers, which can provide the most accurate results but limit the scalability of the approach [142].

4.1.2 Challenges in the Recovery of Function Boundaries

The recovery of function boundaries in a program consists in identifying all function entry points and associating them all with their reachable basic blocks, skipping over function call instructions. This task poses a series of challenges.

First of all, the accuracy of the function boundary recovery is highly dependent on the quality of the underlying CFG. In fact, if, e.g., the CFG lacks information about the destinations of an indirect jump, all the destination basic blocks might not be considered part of the function, leading to a loss in accuracy.

Another issue is deciding whether a certain basic block is the entry point of a function or not. The presence of an explicit function call to its address is a strong indication, but it may not always be available. Specifically, a certain function might never be called directly but only through a function pointer, a C++ virtual call, or a tail call.

Furthermore, several common challenges emerge while trying to identify function boundaries across architectures. In the following we report some of the most relevant:

Call thunks. In ISAs where the program counter is not addressable, it is a common practice to perform a function call to the next instruction so that the program counter becomes available on the stack or in the link register. The destination of such a function call should not be mistakenly interpreted as the entry point of an actual function.

`noreturn` functions. A `noreturn` function, in C terms, is a function that never returns (e.g., `exit` or `longjmp`). These functions are sometimes called through a function call instruction and not through a simple jump. This leads to a spurious path from the call site to the next instruction, which might even be part of a distinct function.

Shared code. Two functions may share a portion of their bodies, in particular, two hand-written assembly functions might share the footer or a sequence of instructions for error handling.

Calls to the middle of a function. In certain cases, a function might have multiple entry points. This case is mostly seen in hand-written assembly and it is usually employed to provide a faster version of a function that does not verify certain preconditions that are known to hold.

Tail calls. Tail calls appear in the code as simple unconditional jump instructions, and, therefore, have to be handled in a way that prevents them from being mistakenly identified as part of the function-local CFG.

4.2 Design

The basic approach illustrated in Chapter 3 recovers (rudimentary) information about the control-flow graph. While the presented approach suffices for cross-ISA binary translation (due to the option to fall back to an “oracle” mapping table that mitigates imprecision in the analysis at run-time), the precision is too low to accurately identify function boundaries.

Recovering an *accurate* control-flow graph is a much more ambitious and challenging process. For this reason, in the first part of this section we substantially extend and increase the CFG analysis to improve its accuracy. In the second part of the section we describe the function boundary recovery process that is only possible on an accurate CFG.

4.2.1 Handling of Reaching Definitions

OSRA propagates tracked values across load/store instructions. Therefore, it is critical to know which definitions (i.e., store instructions) reach a certain load and viceversa. This information is provided by our *reaching definition analysis.*

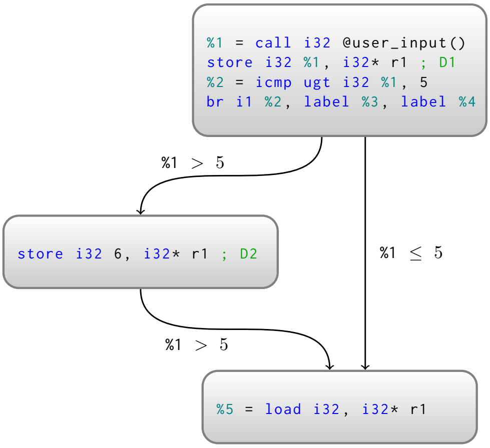
Here, we introduce three extensions to the basic reaching definition analysis used in Section 3.3.3: (i) merging reaching definitions, (ii) path-sensitive merging, and (iii) conditional reaching definitions. We also discuss how these improvements are integrated into OSRA. These extensions improve the number of jump targets recovered from indirect jumps in common scenarios encountered while analyzing binaries of different architectures, increasing the accuracy of the recovered CFG.

Merging Reaching Definitions. One of the limitations of OSRA consists in the loss of precision every time a load is reached by multiple definitions. In this case, the load instruction is associated with a $\top$ value, i.e., a self-referencing, unconstrained OSR. Therefore, all constraints available through the reaching definitions are lost, resulting in an over-approximation that reduces the accuracy of the overall analysis.

Defining a merging policy for the OSRs associated with the reaching definitions addresses this challenge.

Suppose a load instruction is reached by n reaching definitions r_i associated with an OSR in the form $a_i + b_i \times x_i$, with $x_i \in [c_i, d_i]$. All the OSRs of the reaching definitions are considered. The merge is performed only if all the multiplying factors b_i are the same. If this is the case, a_i/b_i is added to the lower and upper bounds (c_i and d_i) of each *bounded value.* All the resulting *bounded values* are then merged according to the *or*-policy (i.e., computing the union of the constraints). If the resulting set of ranges can still be represented with the expressive power of a *bounded value* (i.e., as a single contiguous range, possibly negated), then it is employed to build a new *bounded value* that will be associated with an OSR referencing the load instruction itself. Finally, an OSR, having $a = 0$ and $\forall i, b = b_i$, will be assigned to the load instruction.

Path-sensitive Merging. Even in the presence of the above merge policy, due to how constraints are propagated, some useful constraints known to hold on the path from the definition to the load may not survive until they reach the load. Such a situation can be explained by tracking the reaching definitions of the load



```
%1 = call i32 @user_input()
store i32 %1, i32* r1 ; D1
%2 = icmp ugt i32 %1, 5
br i1 %2, label %3, label %4
```

```
store i32 6, i32* r1 ; D2
```

```
%5 = load i32, i32* r1
```

Figure 4.1: LLVM IR example of the need for *path-sensitive merging*. The branch instruction propagates the constraint `%1` > 5 in the *true*-branch and `%1` ≤ 5 in the *false*-branch. The final basic block receives both constraints, resulting in useless information about `%1`. However, from the perspective of the `load %5`, only the constraint coming from the *false*-branch is relevant, since the store in the *true*-branch aliases the load from $r1$.

`%5` in Figure 4.1. Consider the definition $D1$ that uses `%1`. When the control-flow splits due to a conditional branch we obtain two opposite constraints about `%1` on the successor basic blocks: `%1` > 5 and `%1` ≤ 5. When the two paths later merge again, those constraints cancel each other out. However, we are only interested in the constraint on `%1` along one of the two paths from `%1` to the load `%5`: the one on the *false*-branch. In fact, along the path that goes through the *true*-branch we have $D2$, a definition of $r1$ aliasing $D1$.

To handle such cases, we need to make the merge policy path-sensitive. The key idea of the *path-sensitive merge policy* is to visit, in depth-first order, all the ancestors of the basic block containing the instruction l loading the variable x and stop when a basic block containing a definition d of x is met. When this happens, all the constraints on the path from d to l on the value associated to the OSR of d are considered. First, since all of them have to hold on the path from d to l, they are *and*-merged together, i.e., the intersection of the constraints is computed. Then, the resulting constraint is accumulated in a *result* variable through an *or*-merge policy. Therefore, at the end of the process, *result* will contain a constraint holding on all the paths from each definition of x to the load d.

Algorithm 4.1 details how the *path-sensitive merge policy* is implemented. First we initialize the *result* variable with an empty constraint and create an empty stack s_i for each reaching definition i. Then, another stack ws is created to support our depth-first exploration of the ancestors of the basic block l containing our target load l. ws contains a pair of basic blocks $\langle a, b \rangle$ which are used to identify the edge $b \to a$ that needs to be explored next. ws is initialized with the edge going from l to its first predecessor.

```
100: cmp   r2, #0              1: lt = r2 < 0;
104: addlt r2, r2, #1          2: if (lt) { r2++; }
108: blt   exit                3: if (lt) { return; }
10c: add   r1, r1, r2          4: r1 = r1 + r2;
```

Figure 4.2: On the left, ARM assembly snippet with multiple consecutive instructions sharing the same predicate. On the right, equivalent C pseudo-code.

The algorithm keeps considering the top element of the stack ws until all the paths from the reaching definitions i to l have been explored. In each iteration, the next element to be visited is first recorded on ws and then the $cur \rightarrow origin$ edge is considered.

First of all, all the stacks s_i are reset to the same height as ws. Then, the constraints holding on $cur \rightarrow origin$ on each reaching definition i are pushed onto the respective stack s_i. If cur contains one of the reaching definitions i, all the constraints on the stack s_i are *and*-merged and the result is accumulated in $result$ (with an *or* merge policy). On the other hand, if cur does not contain a reaching definition we can proceed one level deeper in our exploration, and the edge coming from the left predecessor of cur is registered on the stack ws.

If we apply this approach to Figure 4.1, we first observe that $D1$ is expressed in terms of `%1`, therefore its stack s_{D1} will collect constraints on `%1`. $D2$ does not need a stack, since it is a constant definition. We start from the last basic block, proceed to its first predecessor, push `%1` > 5 on s_{D1} and find the definition $D2$. Since the definition is constant, we directly *or*-merge it in $result$, obtaining the constraint `%5` $= 6$. Note that the constraint `%1` > 5 is ignored, since it is not related to the value being stored in the definition $D2$. At this point, we remove an element from the stack s_{D1}, proceed to the right predecessor, push `%1` ≤ 5 on s_{D1} and meet $D1$. By *and*-merging all the constraints on s_{D1} we obtain `%1` ≤ 5, which is in turn *or*-merged into $result$, leading to the final constraint `%1` ≤ 6, as expected.

To keep the algorithm simple, each edge is visited only once. Note that our analysis only considers load instructions accessing a CSV or the address pointed by a CSV plus a constant offset. As a consequence, performing a conservative alias analysis is straightforward. Note also that employing the *path-sensitive merging* policy is resource demanding, and, therefore, we only employ it as a fallback if the straightforward analysis is not successful.

Conditional Reaching Definitions. Depending on how reaching definitions are computed, the accuracy of our system varies. In particular, the naïve implementation may lead to spurious reaching definitions in ISAs heavily employing predicated instructions.

Consider the ARM assembly snippet on the left of Figure 4.2. The `r2` definition at `0x104` (the `add` instruction) should not reach the use at `0x10c`, since, if the addition is executed, the branch gets executed too (they share the same predicate). However, a binary analysis system has to consider each instruction on its own, therefore it will interpret those instructions as illustrated in pseudo-C on the right of Figure 4.2. In this situation, a traditional reaching definition analysis would propagate the `r2` definition in the body of the `if` block to line `3`. Then, since no other definition aliases it, it would not only be further propagated to the body of the second `if`, but also (incorrectly) to the next instruction, reaching the use at line `4`.

For this reason, we created the *condition numbering analysis* (CNA), which groups all conditional branch instructions that share the exact same condition. CNA checks each pair of conditions to verify if they compute the same operation on either the same operands, or on operands reached by the same set of definitions. Such a grouping mechanism is efficiently implemented through a hash-map using an appropriate hash function considering the involved operations and their operands.

The CNA's results are then employed by the *conditional reaching definitions analysis*. This analysis, along with tracking definitions, records whether a condition identified by CNA (or its negation) holds in each basic block.

When a definition is propagated to the successors of a conditional branch, the identifier of the branch's condition is retrieved, and, if present in the set of conditions known to hold in the basic block containing the definition, it is propagated only to the *true*-successor. Otherwise, if the negated condition is present, it is propagated only to the *false*-successor.

In practice, going back to the example in Figure 4.2, the two `lt` conditions are identified by the same integer, say 42. Therefore, when propagating the definition at line `2` in line `3`, the branch condition is inspected, and since it is also identified by 42, the definition proceeds only towards the *true*-branch, preventing it from reaching line `4`.

4.2.2 Function Boundaries Recovery

The function boundary identification process is split into five steps that we present in the following.

1. Identify Call/Return Instructions. As explained in Section 2.2.5, by design, our analyses have to be ISA-agnostic, and therefore we assume that the underlying IR used for the analysis does not explicitly provide the concept of *function call* or *return* instructions. For this reason, we redefined these two concepts in an architecture-agnostic way.

Function call. A branch instruction preceded by an instruction performing a store of a constant integer matching the next PC, considering delay slots if necessary. This integer is the *return address*.

Return. Any indirect branch instruction whose destinations are either unknown or an address known to be a *return address*.

Note that these definitions are generic enough to handle all the known actual implementations of function call instructions in real ISAs. Specifically, the function call definition successfully captures both architectures saving the return address in a register (e.g., `lr` for ARM, `ra` for MIPS) or on the stack (e.g., x86). The first step in the function boundary recovery process consists in scanning the code for instructions matching the *function call* definition, and then, once all the possible *return addresses* have been collected, for instructions matching the *return* definition.

2. Identify Initial Candidate Set. The second step collects an initial set of *candidate function entry points* (or CFEPs). Specifically, we have three initial types of CFEPs.

(a) **Called jump targets.** The most important and reliable source of CFEPs are function calls, since they explicitly indicate that their destinations are functions.

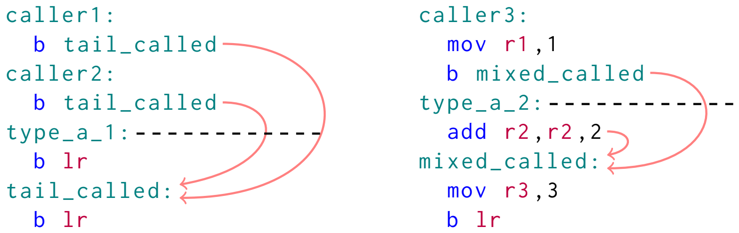
(b) **Unused jump targets in global data.** Global data can also be a source of CFEPs, e.g., due to function pointers stored in global data or C++ virtual tables. However, global data also contains jump tables used to implement C `switch` statements. These addresses do not represent pointers to a function and may lead to a large number of false positives. For this reason, we only consider the *unused* portion of global data. By *unused* we mean that a specific interval in global data has never been accessed by SET. In fact, as mentioned in Section 3.2.2, SET can read global memory areas to materialize addresses contained in a jump table, which are therefore blacklisted.

(c) **In-code constants.** The code itself can contain function pointers, for instance if a function pointer is materialized in a register and then stored to memory. All the jump targets recovered by SET are considered and filtered: we register only jump targets that never end directly in the PC and that are never used as a load address. The rationale behind these choices is that values ending up in the PC will become part of the regular CFG of the program and we can handle them in other ways, while if a load is performed at a certain address, we assume that the target is data, and not code.

We say that a CFEP has its *address taken* if it is of type (b) or (c).

3. Identify Reachable Basic Blocks. Once a preliminary set of CFEPs is available, for each one of them we follow the CFG and associate each basic block reachable from there with the CFEP. When we reach a call instruction we do not follow it, but we proceed to its return address, and when we reach a return instruction, we stop our exploration. Moreover, when associating a basic block with a CFEP, we also keep track of how we reached it, that is either through regular control-flow or by proceeding to the return address of a call instruction.

Once all the basic blocks reachable from a CFEP have been identified, each branch instruction is inspected again to verify if it is a *skipping jump*. A *skipping jump* is a jump instruction that has at least a CFEP of the type (a) between its location and its destination. This check is performed to identify if the branch instruction is jumping over a basic block we reliably know to be the entry point of a function. This type of instructions are often a hint for the presence of a tail call, therefore we create a new CFEPs out of their targets and process them as described.



```
caller1:
  b tail_called
caller2:
  b tail_called
type_a_1:
  b lr
tail_called:
  b lr
```

```
caller3:
  mov r1,1
  b mixed_called
type_a_2:
  add r2,r2,2
mixed_called:
  mov r3,3
  b lr
```

Figure 4.3: ARM example of *skipping jumps*. `tail_called` is a valid CFEP, since it can be reached only through *skipping jumps* (going on over the type (a) CFEP `type_a_1`). On the other hand, `mixed_called` is discarded as a CFEP and its basic basic block is considered part of `caller3` and `type_a_2`. In fact, `mixed_called` can be reached both through a *skipping jump* (coming from `caller3`), but also through the fall-through path after the `add` instruction in `type_a_2`.

4. Filter Candidates. At this point we have sufficient information to perform an evaluation of which CFEPs should be kept, and which should instead be discarded. The criterion to keep or discard a CFEP is expressed with a simple rule:

> *The CFEP is kept if it is reachable exclusively through call instructions or skipping jumps.*

In practice we want to keep all the CFEPs except those that are reachable through the local CFG of another function (i.e., return paths from a function call, fallthrough paths or jumps not going over other CFEPs). This means that we preserve CFEPs whose addresses are taken or are reachable only through tail calls, as long as they do not appear to be part of the local CFG of another function. On the other hand, we discard CFEPs which are reachable both through the local CFG of a CFEP and *skipping jumps*, since this is a strong hint that the *skipping jump* is not a tail call, but simply a result of two functions sharing some code.

Figure 4.3 reports an example of a CFEP reachable only through *skipping jumps* (`tail_called`, on the left) and a CFEP reachable both through a *skipping jump* and function-local control-flow (`mixed_called`, on the right). The latter example, is a typical situation where two functions share the main part of their body but have slightly different headers. In these situations, we deem it appropriate to assign the basic blocks of the main part of the body to both functions.

5. Finalize the Set. The last step consists in promoting all the jumps to the survived CFEPs to the status of function calls and consequently recompute for each one of them the set of basic blocks reachable from the entry point. The final result is a set of functions, possibly sharing code.

`noreturn` Functions. Consider the following ARM snippet:

```
main:
  add     r0, r0, #3
  bl      exit
hello:
  add     r0, r0, r1
  bx      lr
```

Note how the `main` function does not have a return instruction, in fact it is not necessary since the `exit` function will never return. In C terms, `exit` is known as a `noreturn` function.

While exploring the basic blocks reachable from `main`, if our analysis does not identify `exit` as a `noreturn` function, we might mistakenly assign basic blocks belonging to the `hello` function to `main`. Therefore, identifying `noreturn` functions is paramount to accurately recover function boundaries.

We detect the following types of `noreturn` functions:

Syscall wrappers. Before each syscall we inject an instruction loading the CSV associated with the register holding the syscall identifier. In this way, our reaching definition analysis will provide a list of all the reaching definitions. These definitions are monitored by SET, and, in case we notice that one of them writes a constant value corresponding to the identifier of a `noreturn` syscall (such as `exit`), we mark its basic block as a *killer basic block*.

Infinite loops. We mark all the basic blocks belonging to a loop in the CFG of a function with no exit nodes (i.e., an infinite loop) as *killer basic blocks*. Such a situation is typical in the implementation of the `abort` function as a last chance to prevent execution from proceeding, in case raising a signal does not have the desired effect.

`longjmp`. Our analysis also looks for basic blocks that overwrite the stack pointer register with a value that is neither obtained as an offset from its previous value (e.g., `sp = sp + 8`) nor loaded from a memory address relative to its value (e.g., `sp = *(sp - 16)`). Such a behavior typically identifies the `longjmp` function and its derivatives. Such basic blocks are marked as *killer basic blocks* too.

At this point, all the *killer basic blocks* are temporarily modified to have a single successor: the *sink*. All the nodes post-dominated by the *sink* are in turn marked as *killer basic blocks*. In practice this means we reach the entry point of functions such as `abort`, `exit`, `execve`, `longjmp`, and all their wrappers and correctly identify them as `noreturn` functions.

4.3 Experimental Results

The analysis framework and the set of analyses presented in the previous section have been implemented in `rev.ng`.

To evaluate our prototype, we focused on Linux binaries on three popular architectures:

MIPS. Using GCC 5.3.0 and clang 3.8 with musl [128].
ARM. Using GCC 5.3.0 and clang 3.8 with uClibc [54].
x86-64. Using GCC 4.9.2 and clang 3.8 with musl [128].

As in the experimental evaluation of Section 3.4, our choice was guided by the diversity of their features such as register size, presence of delay slots, support of predicate execution, CISC/RISC designs, endianess, and variable-length instruction encoding. To test the robustness of our approach, all the binaries we employed were stripped of debugging information and linked statically.

Note that statically linked binaries provide less information than dynamically linked executables, since the dynamic table and the dynamic symbols are missing. This also means that our tool handles the C standard library, which is large, includes hand-written assembly and other manually optimized pieces of code which are not typically found in programs. Extending our work to support dynamically linked programs consists in loading the main binary and all its libraries and perform our analyses on the whole code corpus. In summary, using statically linked binaries, results in the most challenging setting.

The only "structural" information we left available to the evaluated tools was the section list, which allows distinguishing between code and data regions. This distinction enables, e.g., `rev.ng` to exclude spurious jump targets and function calls which would introduce noise in our evaluation. Section information is preserved even when `strip`ing an ELF binary. Symbols are only employed to collect the ground truth: our tool never uses them to recover function boundaries.

4.3.1 Accuracy of the Recovered Function Boundaries

We built the 105 programs of the coreutils project for the three architectures, including `md5sum`, `ls`, `install`, `df`, `cp`. The programs, including the C standard library, have been compiled using GCC and clang in three different configurations: optimized for performance (`-O2`), aggressively optimized for performance (`-O3`) and optimized for code size (`-Os`). Since uClibc does not support clang, for ARM, the C standard library has been compiled using GCC in all the configurations. Table 4.2b reports the average size of the code section (`.text`) for each tested configuration.

Collecting ground truth for *control-flow graphs* is a challenging task. In GCC, CFG information cannot be obtained, since the back-ends implicitly generate basic blocks by printing strings of assembly. We considered LLVM, but since *each back-end* must be instrumented in non-trivial ways, it would have resulted in prohibitive engineering effort. For this reason, we focused the evaluation on the accuracy of the recovered function boundaries instead. Since an accurate CFG is a requirement for recovering accurate information about function boundaries, the presented results can be considered a *lower bound* for the accuracy of the CFG itself.

The ground truth for function boundaries is easier to recover. Specifically, we employed the `STT_FUNC` ELF symbols from the binaries, which provide the starting address and size for each function. From these ranges we excluded constant pools, since they should not be translated, and `nop` instructions, since they are mostly used for function and instruction alignment purposes.

```
Data: The basic block of the target load ($l$) and the set of basic blocks of its
      reaching definitions ($d = \{d_i\}$).
Result: The merged constraint $result$.
$result = \bot$;
create an empty constraints stack $s_i$ for each reaching definition $i$;
create a stack $ws$ of $\langle basic\,block, basic\,block \rangle$ pairs;
$ws$.push($\langle l$, firstPredecessor($l$)$\rangle$);
while ws is not empty do
    $\langle origin, cur \rangle = ws$.pop();
    foreach reaching definition i do
        cut $s_i$ to the height of $ws$;
        $c$ = getConstraint($i, origin, cur$);
        $s_i$.push($c$);
    if cur is not the last predecessor of origin then
        $ws$.push($\langle origin$, nextPredecessor($origin, cur$)$\rangle$);
    $stop$ = false;
    foreach $d_i$ in $d$ do
        if $d_i = cur$ then
            $tmp = \top$;
            foreach constraint $c_k$ in $s_i$ do
                $tmp = tmp$ and $c_k$;
            $result = result$ or $tmp$;
            $stop$ = true;
    if not stop then
        $ws$.push($\langle cur$, firstPredecessor($cur$)$\rangle$);
return $result$;
```

Algorithm 4.1: The *path-sensitive merging* algorithm for constraints. firstPredecessor(a) returns the first predecessor of basic block a, nextPredecessor(a, b) returns the next element in the list of predecessors of a after b, while getConstraint(i, b, c) returns the constraint on the i-th reaching definition holding on the edge $c \to b$ (i.e., from basic block c to basic block b).

			x86-64				MIPS			ARM			
			IDA	`rev.ng`	BAP	angr	IDA	`rev.ng`	angr	IDA	`rev.ng`	BAP	angr
Jaccard index	GCC	`-O2`	97.98	98.00	84.47	90.92	96.37	94.30	84.95	96.56	95.60	79.10	67.13
		`-O3`	98.35	96.72	85.31	91.06	95.17	93.16	82.94	96.41	95.60	79.08	66.65
		`-Os`	98.16	98.77	87.03	93.08	97.10	94.79	88.82	95.92	94.08	81.67	66.38
	clang	`-O2`	98.39	98.31	85.84	92.05	96.41	92.48	78.50	94.62	94.49	78.34	65.98
		`-O3`	98.34	97.77	82.73	91.90	96.31	91.65	78.22	94.64	94.53	78.16	63.43
		`-Os`	97.83	98.44	83.70	91.01	96.52	92.66	81.07	94.01	93.38	80.84	65.89
Matched (%)	GCC	`-O2`	94.56	98.37	83.51	93.76	93.09	98.13	93.32	85.59	88.77	80.31	97.27
		`-O3`	93.25	98.43	83.30	94.59	92.33	97.58	95.44	84.18	87.90	79.35	97.47
		`-Os`	95.31	97.90	83.39	93.15	92.60	97.03	93.21	88.22	87.82	78.23	97.37
	clang	`-O2`	94.31	98.83	67.42	94.29	84.91	83.26	78.26	84.01	86.92	77.37	97.20
		`-O3`	94.27	98.78	59.79	94.64	85.66	83.35	78.33	83.45	86.38	77.46	94.41
		`-Os`	94.91	98.65	59.20	93.85	86.35	84.50	79.06	86.91	85.65	76.87	96.60
RSS (MiB)	GCC	`-O2`	227.88	248.59	284.62	297.63	195.86	628.95	201.24	198.62	463.88	264.71	261.58
		`-O3`	228.67	289.81	286.67	297.03	197.72	4325.39	1017.87	199.01	568.37	280.59	273.94
		`-Os`	227.08	229.48	261.51	256.15	196.02	784.30	223.17	198.24	369.03	235.64	259.80
	clang	`-O2`	226.67	215.32	200.39	213.04	196.63	549.36	233.53	198.73	401.64	268.06	244.27
		`-O3`	227.07	236.36	177.09	204.82	196.19	584.53	207.99	198.99	439.32	282.41	508.11
		`-Os`	226.47	743.53	149.52	734.89	196.06	574.46	224.67	198.29	1293.56	245.84	676.03

Table 4.1: Comparison of the experimental results obtained by IDA Pro, `rev.ng`, BAP, and angr on the 105 coreutils binaries compiled using GCC and clang for x86-64, MIPS and ARM. Three different optimization levels have been employed: `-O2`, `-O3`, and `-Os`. Note that BAP does not support MIPS. The *Matched* rows represent the percentage of functions that have been matched at least partially, ignoring the quality of the match. *Jaccard index* is the average Jaccard index of each detected function against its best match in the set of original functions. The average is weighted over the size of the function and over the size of the `.text` section of each program. Finally, *RSS* is the average (over all the binaries) of the maximum resident set size (i.e., peak memory usage).

We compare our results against related work by letting the tools produce a CFG. Then, starting from each entry point, the CFG is explored, and each basic block reachable from there is recorded as part of that function. Since one of our goals was to assess the quality of the CFG, we ignored the basic blocks that were assigned to a function but that were not reachable from the entry point, since this means that the tool assigned a basic block to a function but could not understand how it takes part of the CFG of the function.

We compared `rev.ng` with IDA Pro 6.6 using a custom IDAPython script, the most recent version of angr (as of November 2016) employing the `CFGFast` CFG recovery option [142], and BAP 0.9.9, which implements the ByteWeight approach [11], with the `--phoenix` option and collecting the CFG data from the GraphViz output. We employed the latest available ByteWeight signatures and we extended BAP to output the size of basic blocks. Note that BAP does not support MIPS.

Table 4.1 reports the results of our experiments. The most important information needed to assess the quality of the results is the *Jaccard index*, which we computed for each detected function against its best match in our ground truth. The index is computed comparing the set of the basic blocks assigned to the function against the actual set of basic blocks, according to the ground truth. The Jaccard index provides a concrete measure of the accuracy of the match, penalizing missing or extra basic blocks. Table 4.1 reports another interesting metric reported in the *Matched* column, that is the percentage of matched functions, ignoring the quality of the match.

The results in terms of accuracy of `rev.ng` are very close to those of IDA Pro, and sensibly better than those of BAP and angr in all the tested configurations. The difference between IDA Pro and `rev.ng` comes from few functions that only IDA Pro identifies. By performing manual inspection of the functions detected by IDA Pro but not by `rev.ng`, we verified that in most cases it is dead code, i.e., code whose address is not *taken* and has no direct control-flow transfers pointing to it. This is due to the fact that the heuristics implemented in IDA Pro can detect function prologues. However, since in all inspected cases the difference was due to dead code, we do not consider this a limitation but a design choice.

Note that we already tried to mitigate this problem by compiling our code using the `-ffunction-sections` GCC option, along with `-Wl,--gc-sections`, which is supposed to minimize the amount of dead functions. However, hand-written assembly functions are not pruned.

In addition to dead code, we found additional sources of inaccuracy in the CFG, which affect both `rev.ng` and IDA Pro:

Aggressively optimized nested `switch`. Under certain conditions, in x86-64 functions using nested `switch` statements, `rev.ng` was unable to track the size of the jump tables used by the inner `switch` statement. Since the starting address of the jump table was available, we could devise a heuristic to recover it. However to provide coherent results, we decided not to do so.

Jump table addresses spilled on the stack. In certain situations, in MIPS in particular, GCC can spill the starting address of a jump table on the stack in the

	x86-64	MIPS	ARM
IDA	5.17 s	8.06 s	5.39 s
rev.ng	44.57 s	222.11 s	119.53 s
BAP	35.33 s	n.a.	25.48 s
angr	384.15 s	273.83 s	147.82 s

(a)

		x86-64	ARM	MIPS
GCC	-O2	115.12 kiB	98.50 kiB	115.12 kiB
	-O3	161.12 kiB	105.01 kiB	161.12 kiB
	-Os	123.81 kiB	89.11 kiB	123.81 kiB
clang	-O2	138.72 kiB	102.56 kiB	138.72 kiB
	-O3	125.51 kiB	106.03 kiB	125.51 kiB
	-Os	126.59 kiB	95.12 kiB	126.59 kiB

(b)

Table 4.2: Table (a) reports the time spent (in seconds) to collect the control-flow graph and the function boundaries of the `ls` binary, compiled with `-O2` using GCC for x86-64, MIPS and ARM. The presented results are averaged over 10 runs. Table (b) reports the average size (in kilobytes) of the `.text` section of a coreutils program compiled using the specified configuration.

function prologue because it might be used multiple times across the function. While `rev.ng` implements a basic mechanism to track stack values, doing so across function calls is non-trivial, since during the CFG recovery phase we have no information about function calls.

In our (non-exhaustive) exploration, we did not find major inaccuracies in the CFG that IDA Pro handled and `rev.ng` did not. On the contrary, we found several examples where `rev.ng` is more precise than IDA Pro. The next section discusses such an example as a case study.

Looking at BAP's results, we found that (i) some functions were missing several instructions in the function prologue and (ii) a series of spurious functions in the middle of actual functions manipulating the stack (e.g., for variable-length arrays). Tail calls and indirect jumps due to `switch` statements also appeared to be handled poorly.

For what concerns angr, despite often matching the most functions compared to the other tools, the accuracy of the matching is lower. The main issues are related to mishandled predicated return instructions, code after `noreturn` function calls forced to a new function in all cases, and incomplete handling of certain indirect jumps due to `switch` statements. It is worth noting that the angr project was hand-tuned for the x86 architecture, which we did not evaluate.

In conclusion, `rev.ng` results are comparable or sensibly better compared to the other evaluated tools, proving the effectiveness of our approach.

Memory Usage and Processing Time. The last set of rows in Table 4.1 reports, for each tool, the peak memory usage (*RSS* column) averaged over all the 105 processed binaries. As the table shows, `rev.ng` memory usage is mostly comparable to other tools on x86-64, but on ARM and MIPS our tool results to be more resource demanding. This is due to a limitation of our current implementation. Specifically, OSRA propagates each constraint indefinitely, leading to a considerable memory overhead. Our development branch addresses this issue by limiting the propagation of constraints on a SSA value up to the farthest instruction employing it in an OSR. In our preliminary testing of such a solution on the `ls` binary, the memory consumption is reduced from 1.78 GiB to 899 MiB on MIPS and from 1.19 GiB to 476 MiB on ARM.

Since the continuous integration system where we run our tests is composed by servers featuring different hardware specifications, Table 4.1 does not report timing results. Instead, we collected timing results on a single machine with 32 GiB of RAM and an Intel i7-6820HQ CPU, with 4 physical cores clocked at 2.7 GHz. We ran each one of the four tools 10 times against the `ls` binary compiled for MIPS, ARM and x86-64. Table 4.2a reports the results. As expected, tools such as IDA Pro and BAP, which employ heuristics or machine learning techniques, are notably faster compared to `rev.ng`. On the other hand our prototype implementation outperforms angr. Note however that the detailed information we collect can serve as a basis for more sophisticated analyses whose purposes goes beyond recovering the CFG or the function boundaries. In fact, if we compare the time taken by the IDA Pro's Hex-Rays Decompiler to analyze the whole binary, we get more comparable results. In particular, the Hex-Rays decompiler took approximately 37 s to analyze `ls` compiled for x86-64 and 27 s for the ARM version. Note also that the previously mentioned development branch of `rev.ng` reduces the analysis time for `ls` from 44.57 s to 31.59 s on x86-64, from 222.1 s to 15 s on MIPS and from 119.53 s to 50.53 s on ARM.

4.3.2 Case Study: the Buggy Memset

Figure 4.4 shows a simplified version of the ARM `memset` implementation included in uClibc [54]. It is a hand-optimized implementation that copies 8 bytes at a time (see the `copy_loop` label), and then copies the (at most 7) leftover bytes one by one (see the `remaining` label).

We consider the recovery of the CFG of this function interesting for several reasons. Specifically, it would be beneficial to prove that instruction `Z` can only reach one of the 7 `strb` instructions or the return instruction. As we will see, most of the analyses presented in Section 4.2.1 have to be employed.

First of all, `r2` cannot be expressed in terms of a single value. In fact, its usage in `Z` can be affected by the definition in `C`, `E` or any other definition of `r2` in the callers of `memset`. A merge policy with multiple reaching definitions must therefore

```
memset:
copy_loop:
  cmp     r2, #8              ; A
  blt     remaining           ; B
  ; ...
  sub     r2, r2, #8          ; C
  cmp     r2, #8              ; D
  subge   r2, r2, #8          ; E
  bge     copy_loop           ; F
remaining:
  add     pc, pc, r2, lsl #2 ; Z
  nop
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  strb    r1, [r3], #1
  mov     pc, lr
```

Figure 4.4: uClibc ARM implementation of `memset`. `r2` contains the size of the buffer. `copy_loop` is a (partially omitted) unrolled loop copying 8 bytes at a time, while `remaining` takes care of copying the bytes left over by `copy_loop`. Each one of the 7 `strb` instructions copies a single byte; "`add pc, ...`" jumps to one of them depending on how many bytes are left to copy.

be leveraged.

Second, to have an accurate set of definitions of `r2` reaching `Z`, we need to handle predicate instructions correctly. If the naïve reaching definition approach is employed, the definition in `E` is propagated on both successors of `F`, effectively preventing the definition in `C` from reaching `Z`.

Third, if the adopted merge policy is not *path-sensitive*, the `r2` use in `Z` sees two constraints on the `r2` definition in `C`: `r2`$<$ 8, through the path `CDZ`, and `r2`$\geq$ 8, through the path `CDEFABZ`. This makes the definition of `r2` in `C` unbounded in `Z`, preventing the analysis from proving that `r2` is lower than 8 in all cases. However, using the proposed *path-sensitive* merge policy, the constraint `r2`$\geq$ 8 is ignored: going backward through the `CDEFABZ` path, the definition in `E` prevents the analysis from reaching `C` and taking into account the constraint associated to that path. In conclusion, `rev.ng` was able to correctly recover all the jump targets, unlike all the other tools we tested.

The last reason why this `memset` is relevant consists in a bug it contains and that we discovered through `rev.ng`. All the comparisons performed by this function are signed. Therefore, a malicious user in control of the size parameter of `memset` (`r2`) can control the program counter by simply passing a negative number. This bug was recently fixed in uClibc-ng [30].

4.4 Conclusions

Recovering CFG and function information from binaries is an important technology that enables further analyses like static binary instrumentation, security analysis, reverse engineering, or retrofitting defense mechanisms.

We design a set of analyses that statically recover the CFG and the function boundaries of a binary with high accuracy, without relying on ISA-specific heuristics. While heuristics can be very effective (as shown by IDA Pro), they are not portable to other architectures and require vast manual effort. Developing ISA-independent analyses simplifies (or even removes) any porting efforts while retaining overall comparable results. Such approaches meet the sweet spot between black box analysis employing machine learning and hard-coding ISA-specific heuristics.

Chapter 5

Function Prototype Identification

Once an accurate CFG and function boundaries have been recovered, the natural next step is the identification of the functions' interface, i.e., the arguments and return values list.

In this chapter we present a series of analyses aimed at identifying the number and location of arguments and return values in a conservative and ABI-agnostic way. This means that no assumptions about the calling convention are made. As in many other analyses in `rev.ng`, the idea is to develop analyses general enough so that any architecture can benefit from them.

To accurately identify arguments and return values, we build our analyses on the top of a data-flow analysis analyzing the usage of the stack. The data-flow problems will be employing Monotone Frameworks (Section 1.5). Such analysis is also employed to increase the accuracy of the function boundaries detection and handle tricky situations, such as outlined function prologue and epilogues.

Note that this chapter only presents the design of our function prototype identification technique. Due to time constraints it has not been possible to perform a thorough evaluation.

5.1 Problem Statement

In this section we will provide an overview of the main features that differentiate a calling convention from another. Studying such features is key to be able to develop analyses that are flexible enough to take into account all the diversity of the ABIs in the wild.

Then we will focus our attention to a specific optimization technique widely employed in embedded architectures to reduce the code size: code outlining, and function prologue/epilogue outlining in particular. Such an optimization can lead to identify *support* functions artificially generated by the compiler as actual functions, while their code should be considered part of the caller functions. Since these support functions often manipulate the stack, their misidentification as standalone functions can compromise the accuracy of any subsequent analysis relying on the correctness of the information provided by the stack analysis. It's therefore vital to be able to identify them correctly, without falling back to *ad hoc* handling.

5.1.1 Calling Conventions Overview

In the following, we list the main features that differentiate one calling convention from the other.

Link register. Each ISA provides, in some form, a `call` instruction. A *call* is basically a normal jump instruction which also stores the return address somewhere. Many RISC architectures, store the return address in a dedicated register (usually known as the *link register* or *return register*), while other architectures, such as x86, store it on the top of the stack. Symmetrically, the `return` instruction performs a jump either to the content of the *link register* or pops into the program counter the top of the stack. Note that in case the *link register* is employed, all the non-leaf functions need to explicitly spill it on the stack and restore it before returning to the caller.

Return values. All the architectures we considered store the function return values in one or more registers. If the return value is too large to fit in the dedicated registers (e.g., a large `struct`), then the function is transformed to return `void` and an extra argument is injected in the function prototype containing a pointer to a caller-allocated buffer of the size of the original return value where the callee is supposed to write the return value.

```
big_t func(void) {                   void func(big_t *R) {
  big_t Result;                        /* ... */
  /* ... */              ⟹             *R = Result;
  return Result;                     }
}
```

Another key observation about return values is the fact that in certain architectures the set of registers for function arguments overlaps those employed for return values. For example, in ARM the registers `a1` and `a2` are used both as function arguments and return values. This means that the empty function could be implicitly forwarding its two first arguments as return values.

Caller/callee-saved registers. Each calling convention divides the general purpose registers into two categories: *caller-saved* and *callee-saved* registers. *Caller-saved* registers are registers that are not guaranteed to be preserved across function calls. Therefore, if a function needs to reuse the content of such a register after a function call, it has to store it on the stack and restore it after the call. Since performing this operation has a cost, it's rarely done and *callee-saved* registers are preferred for such an use case. For this reason, *caller saved* registers are also known as *scratch registers*: they are more fit for temporary computations that can be discarded before the next function call.

On the other hand, *callee-saved* registers are registers that need to have their initial value on exit. This can be done either by simply not using them, or by spilling them on the stack in the function prologue and restore them in the epilogue. We define a register saved in the latter way an *explicitly callee-saved* register.

Arguments. Typically, the first couple of arguments are passed through registers, and the remaining ones go on the stack. x86 is a notable exception. In fact, in many calling conventions, all the arguments are passed directly on the stack. In case stack arguments are employed, most calling conventions require the caller to cleanup the stack space it has reserved. A notable exception is the so called *PASCAL* calling convention, largely popular on x86/Windows: in this case the callee is responsible for cleaning up stack arguments before returning.

Note that multiple arguments can be packed into a single register argument, and get selected through bit fiddling. The handling of variadic arguments varies wildly among architectures (Section 7.2.2) and it's outside the scope of this work.

We also observed that no ABI uses callee-saved registers for return values. In fact, this wouldn't make sense, since it's not possible to return a result in a register and at the same time preserve its original value. On top of this, no ABI uses callee-saved registers for arguments. This is also reasonable, since function arguments are often the result of some temporary computation that's unlikely to be needed at the return of the call.

5.1.2 Code Outlining

Code outlining is a compiler optimization technique that looks for strings of assembly instructions that appear multiple times in a certain module and factor them out into a function. The original functions will replace them with a call to the *outlined* function. Such an optimization allows to reduce the code size and have frequently used instruction strings hot in cache.

While outlining is a technique applicable to generic instruction strings, certain compilers perform it in a more *ad hoc* fashion for parts of the code which are know to benefit from outlining. The most relevant example is the outlining of the function prologue and epilogue: a very large portion of the functions of a program will need to spill (restore) certain registers at the entry (exit) of a function. Therefore, this sequence of `push` (`pop`) instructions can be moved to an independent function which will act as an outlined function prologue (epilogue). This is particularly beneficial in ISAs where only a single register can be spilled in a single instruction. For instance, this wouldn't make sense in ARM, where many registers can be pushed (popped) in a single instruction.

Listing 5.1 reports example of outlining of function prologues and epilogues found in a QUALCOMM baseband (Hexagon architecture [76]) and in the Intel ME [78] code (ARC architecture) [77]. Note that while the two pieces of code seem equivalent, the former outlined prologue does not manipulate the stack pointer, while the latter does.

While outlining function prologues and epilogues is beneficial from a code size and performance point of view, it's detrimental from a reverse engineering point of view. In fact, having *support* functions that move the stack pointer make the

```
function:
  allocframe(#32)
  call prologue
  ...
  jump epilogue

prologue:
  memd(r30 - 24) = r21:20
  memd(r30 - 16) = r19:18
  memd(r30 - 8) = r17:16
  jumpr r31

epilogue:
  r21:20 = memd(r30 - 24)
  r19:18 = memd(r30 - 16)
  r17:16 = memd(r30 - 8)
  deallocframe
  jumpr r31
```

```
function:
  push blink
  bl prologue
  ...
  b epilogue

prologue:
  push r15
  push r14
  push r13
  j [blink]

epilogue:
  pop r13
  pop r14
  pop r15
  pop blink
  j [blink]
```

Listing 5.1: Examples of prologue and epilogue outlining in Hexagon (left) and ARC (right). Both architectures employ the *link register* (`r31` and `blink`). For what concerns Hexagon, `allocframe` pushes on the stack the frame pointer and link registers, then it updates the frame pointer to the new stack value. `deallocframe` pops them from the stack into the respective registers. `memd(a) = r0:r1` is writing the values of `r0` and `r1` at address `a`. On the other hand, for ARC, `b` is a simple branch instruction, `bl` a function call, and `push`/`pop` have their classic behavior.

analysis non-straightforward. This is an issue, since, in a normal situation, each function should have its own stack frame, and shouldn't write in the caller's stack frame, unless a pointer to it is passed as an argument. To the best of our knowledge, this problem has never been tackled in a non-*ad hoc* way, if at all.

5.2 Design

This section will present the set of analyses that we developed to accurately detect the boundaries of a function and the number and location of its arguments. We will first list the assumptions we made, justifying their reason to be and their limitations. Then we will present the *stack analysis*, which tracks how the stack (and the CPU state in general) evolves in a function and detects the bounds of functions and which registers are *explicitly callee-saved*. Next, we will present a set of analyses aiming at collecting (conservatively) as much information as possible on the register and stack arguments/return values for each function and call site. Finally we will see how we combine all the collected information to produce a final output that can discard all the details and is maximally useful to the end user.

5.2.1 Assumptions

In the following we list the assumptions we made about the code we're analyzing during the design of our analyses. Some of them have been introduced as a consequence of our survey of real world calling conventions (Section 5.1.1) and some of them have been introduced for simplicity reasons.

While the assumptions of the latter type might result in an loss in accuracy and correctness, we are able to evaluate how often they hold in real world programs. In fact, since our framework preserves the semantics of the code being analyzed, we can recompile and instrument it with assertions and counters to check if we end up in a situation violating our assumptions.

A1. No indirect accesses to stack arguments. For simplicity reasons, we assume that stack arguments are only accessed through direct memory operations. In practice if a `struct` containing an array is passed as an argument on the stack, and its elements are accessed through an index that we cannot statically resolve, we lose track of it. Note that this is a different situation from receiving an array or a pointer as an argument, which is a situation we intend to handle.

A2. No dead code. We assume there's no dead code in the program. We make this assumption due to the fact that, as we will see, we detect situations where, e.g., a certain write to a register is either a dead code or an argument initialization. A typical example is having a dead assignment to a caller-saved register (i.e., a write to a register whose value is not read by any instruction) before a function call. These kind of assumptions hold as long as at least a minimal set of optimizations have been run on the program. If the analyzed program is not optimized at all, some of our analyses can be disabled. However, we deem the recovery of arguments in a non-optimized program less interesting and realistic.

A3. No uninitialized reads. In certain cases, due to some undefined behavior in the original C program, the compiler might try to read the value of a registers which has not been initialized. This is a sign of a mistake by the programmer, and for simplicity reasons we ignore this situation. In practice such a situation might end up producing a spurious argument, which, despite being wrong with respect with what the program was supposed to do, it's coherent with what the program will actually do.

A4. No return values on the stack. We assume that return values are never passed back to the caller through the stack. In fact, to the best of our knowledge, this never happens. As we saw in Section 5.1.1, when returning a large struct, there's no real return value, but simply an extra argument representing a pointer to what is, in practice, a local variable of the caller. Such situations can be easily captured by post-processing the results our analyses provide.

A5. Callee-saved registers are not arguments or return values. As noted in Section 5.1.1, it's very unlikely to have arguments or return values go through registers which are also callee-saved. This will allow us to handle them in a special way in a first phase, and ignore them in the more in-depth analyses.

A6. Stack arguments are not preserved. A stack argument is, to all practical purposes, something that concerns the callee. Therefore, we assume that the caller cannot assume that the value of a stack argument will be preserved upon return.

5.2.2 The Stack Analysis

The *stack analysis* is an interprocedural data-flow analysis we developed to track, within certain limits, how the stack evolves and it's used within a function. Our final goal is to precisely classify each branch instruction in a way that allows us to accurately draw the boundaries of a function. Additionally, we also aim to identify *explicitly callee-saved registers*, whose detection is vital for arguments detection.

While providing an in-depth description of the stack analysis is outside the scope of this work, it's important to know that it can handle sophisticated CFGs and call graphs, including *indirect function calls* (i.e., calls whose target cannot be statically determined) and recursive functions.

In practice, the stack analysis aims at tracking, at each program point, the state of the program, including the CPU registers, the stack of the current function and of all of its callers and the global data (where global variables reside). To do so, we analyze all the instructions writing and reading registers and all the load/store instructions. We deal with registers, portions of the stack and global data in a unified way in terms of *slots*, i.e., a register or a portion of data starting at a certain absolute address or at a certain offset within a certain stack frame (e.g., the stack frame of the current function). *Slots* have no size associated to them.

In the stack analysis, every value is an address. An address can be expressed as an hypothetical base address plus an offset. For instance the first slot of the current function's stack frame is identified as `SP0+0`, the stack at offset 8 in the caller's stack frame as `SP1+8`, the global variable at the absolute address 0x1234 as `GLB+0x1234`. Registers have an address too: each register is assigned a unique identifier which acts as an offset. For instance, supposing `rax` has 1 as an identifier, it will be represented as `CPU+1`. Note that even literal constants are addresses, for instance the constant 4 is is seen as `GLB+4`. Since we assume that it's not possible to reach an address relative to `SP1` from an address relative to `SP0` (or `GLB`), we define `GLB`, `CPU`, `SP0` and `SP1` as *alias domains*.

For each *slot* we keep track of two pieces of information: 1) what was the last value being *directly* stored in the slot and 2) what is the last value that *might have been stored there indirectly*.

For what concerns what has been directly stored in the slot last time, we have two types of values: 1) the exact address of a slot, or 2) any address within a certain set of alias domains. The former option is more informative, while the second is more conservative and helps us to handle the uncertain cases.

On top of this, we can also say that a *slot* contains the same value a certain slot had at the entry of the current function, no matter what it was.

For what concerns what *might have been stored indirectly* in a slot, we don't keep track of specific addresses, but simply of the fact that a certain address within a set of alias domains might have been stored there.

Listing 5.2 reports an example of some of the above mentioned situations.

Such an approach keeps the analysis feasible and provides us with sufficient information to achieve the above stated goals.

Branch instruction classification. The stack analysis provides sufficient information to identify the type of a certain branch. Is it part of the regular control flow of the function? Is it a function call? A return instruction?

```
function:
  mov [rsp+0],rbx
  mov [rsp+4],42
  mov rbx,rsp
  add rcx,rbx,rax
  mov [rcx],43
  mov rsi,[rsp+0]
  ret
```

Listing 5.2: Example of how the *stack analysis* tracks the values of a slot. `rbx` will contain an exact address (`SP0+0`), while, assuming the value of `rax` is not available statically, `rcx` will be any address within the `SP0` alias domain. On the other hand, the last direct store to the slot at `SP0+4` is 42 (i.e., `GLB+42`), while the last value that *might have been stored there indirectly* (due to `mov [rcx],43`) is any address within the `GLB` alias domain. Finally, we will be able to state that `rsi` contains a value equal to the one that `rbx` had the entry of the function, no matter what it was.

Before we present the possible types of branch instructions, it is important to introduce our own definition of *function*.

Function. A *function* is a set of connected basic blocks terminated by an indirect jump to the return address (either in the link register or on the top of the stack) and where the stack has an height equal to or lower than the one it had at the entry of the function.

Note that we talk in terms of *stack height*. Assuming the stack grows towards lower addresses (as it's typical), this means that the stack pointer must have the same value it had, or a greater value. A sane function should always restore the stack pointer at its original position, except in the case of the PASCAL calling convention, where the stack could be lower (see Section 5.1.1).

This definition is important since it allows us to capture the idea that outlined function prologues and epilogues (see Section 5.1.2) are not real functions. Specifically it allows us to say that "functions" that alter the stack leaving it in an irregular state (i.e., higher than what it was at the entry of the function) are not real functions, but most likely support functions which should be considered part of their callers for all practical purposes. We define such functions *fake functions.* An example of such a function is the `prologue` function of the ARC example in Listing 5.1. We also define calls and returns from such functions *fake function calls* and *fake returns.*

When the analysis detects that a branch instruction jumping to the return address (i.e., either the link register or the value at the top of the stack) does not leave the stack pointer register in an appropriate position, we abort the analysis of the current function and mark it as a fake function. If necessary, the analysis of the caller function will be resumed keeping into account the fakeness of said function.

With this idea in mind we can classify the branch instructions according to the following list.

Function local. This branch is part of the regular control flow of the current function.

Function call. This instruction is performing a call to a function that has not been marked as fake (so far). If this function has never been analyzed in the past, the analysis of the current function is suspend and a new analysis is spawn for the callee.

Return. This instruction is jumping either to the *link register* or the address at the top of the stack. Moreover, the value associated by the analysis to the stack pointer corresponds to a value that is equal to or lower than its initial value. Note that we automatically detect for each function if the *link register* is used, or if the return address is on the top of the stack by analyzing all the callers of the current function. In case no callers are available, we assume the most popular situation in the current binary.

Fake function call. A call to a function that has been previously marked as *fake*. The analysis of the current function is not interrupted: the callee code is treated as if it was part of the calling function. The return address is recorded: once the fake function will return, the analysis will resume from there.

Fake return. The current instruction is a return instruction, but we're in a *fake* function. The analysis will proceed from the previously recorded return address.

Indirect function call. A function call whose target cannot be determined statically. The worst (i.e., the most conservative) assumption is made about its behavior.

Indirect tail call. An indirect jump instruction whose target cannot be statically determined, but in a context where the stack pointer has a value that is equal to or greater than its original value. While we can't be sure, this is an indication that this jump might be an indirect tail call. This constraint is introduced since a tail call can be performed under the same conditions to which a return instruction can be performed.

`longjmp`. An indirect jump instruction whose target cannot be statically determined, but in a context where the stack pointer has *not* a value that is equal to or greater than its original value. This means that an indirect jump is being performed disregarding the status of the stack, basically leaving it in a broken state. Such a jump cannot be a tail call, but it must represent some low-level system routine fiddling in unusual ways with the state of the program. Since the most relevant example of such a routine is the `longjmp` function, this situation is labeled after it.

Killer. The branch terminates a *killer* basic block (Section 4.2.2). This is an indicator that the current function has at least one path from which it is not possible to return.

The analysis results of basic blocks ending with a (proper) return instruction are merged (i.e., combined) into a sink variable which will represent the summary of the stack analysis of the current function.

Being able to classify branch instruction in a such fine-grained fashion allows us to more precisely define the boundaries of functions, compared to Chapter 4. In fact, while before we were, e.g., assuming that all indirect jumps we cannot track are return instructions, in this case, we have an accurate definition of function and return instruction which we can verify.

Note however that plain tail calls, at this stage, are ignored and inlined into the caller. We plan to handle them through a post-processing phase applying criteria similar to those employed in Section 4.2.2.

Explicitly callee-saved registers. As previously said, the *stack analysis* can say that a certain slot contains the initial value of a certain other slot. Such an information allow us to detect *explicitly callee-saved registers*. In practice, if we find out that, on all basic blocks ending with a return instructions, the last value stored in a certain register corresponds to its initial value, we say that such a register has been explicitly saved.

Note that, by considering only the last direct store, we actively ignore all the indirect stores. While this introduces an approximation (and therefore, false positives), we deem it unlikely to have a situation where this can happen. However, as already mentioned in Section 5.2.1, we can assess how much this happens in real world programs by recompiling the program with appropriate instrumentation.

5.2.3 The Arguments Analyses

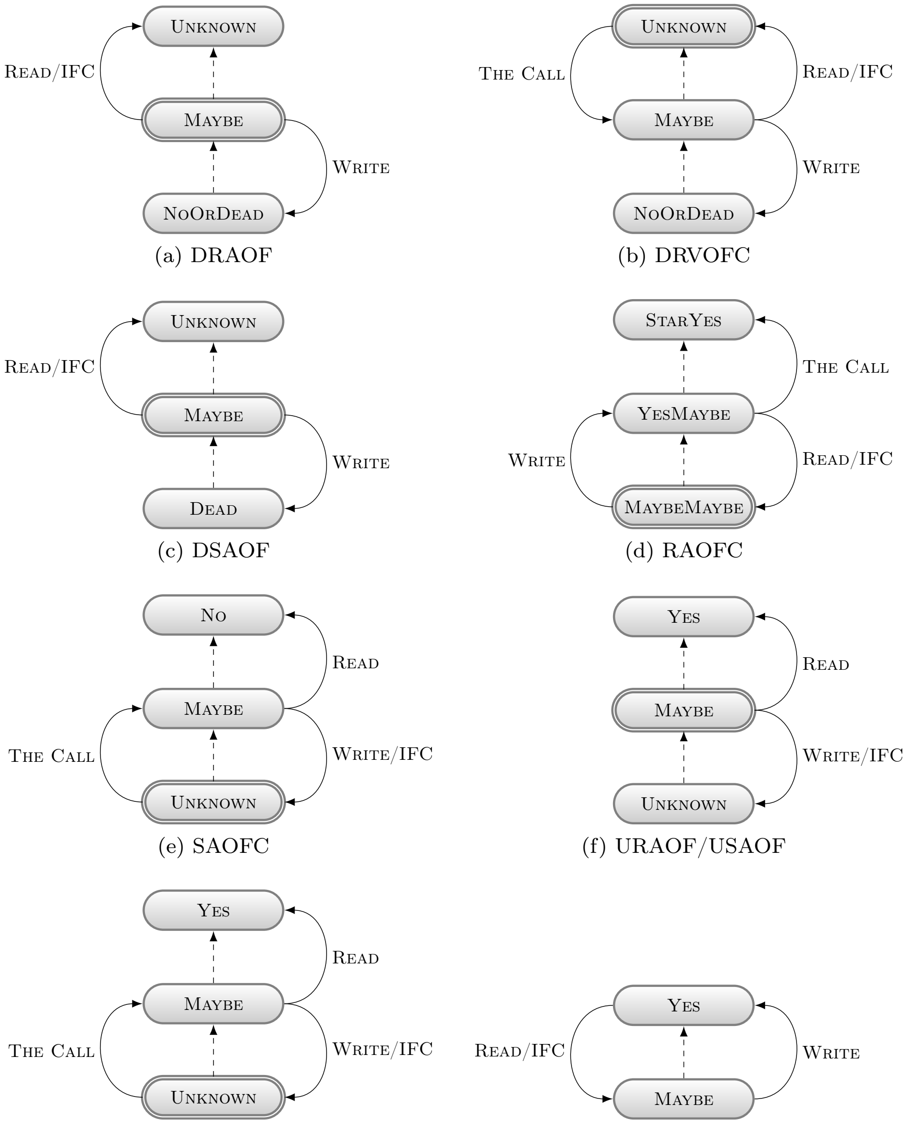
In this section we will present the set of analyses for the detection of arguments and return values and their location. All of these analyses make no assumptions about the ABI (nor the underlying architecture) and explicitly ignore callee-saved or unused registers. Their handling will be introduced in the next section.

Note also that we ignore the fact that multiple arguments can be packed into a single one (Section 5.1.1), since they are in principle indistinguishable from a single argument whose higher and lowest bits are being used independently. These situations can be tackled through the usage of heuristics by post-processing the result provided by our analyses. However this is outside the scope of this work.

We developed eight distinct analyses. Half of them focus on arguments and return values of *functions*, and half of them on arguments and return values of *function calls* (i.e., call sites). Three of them provide information on return values, and the remaining 5 on arguments. Two of them focus on stack arguments, five of them work exclusively on registers and the remaining analysis works both for stack and register arguments. Note that the three analyses focused on return values work exclusively on registers, as per Assumption A4.

Dead Register Arguments Of Function (DRAOF). It detects if a register is in the state "*if an argument, it's dead*". This means that the register is either "*not-an-argument*" or, in case it is, it's dead. Note that we consider an argument to be dead only if it's unused on all the paths of the function. To detect such a situation, DRAOF, checks if, over all paths, the register is clobbered before being read. See Figure 5.1a.

Dead Return Values Of Function Call (DRVOFC). It detects if a register, after a specific function call, is in the state "*if a return value, it's dead*". Note that we consider a return value to be dead only if it's unused over all the paths originating in the function call. To detect such a situation, DRVOFC, checks if,



(a) DRAOF

(b) DRVOFC

(c) DSAOF

(d) RAOFC

(e) SAOFC

(f) URAOF/USAOF

(g) URVOFC

(h) URVOF

Figure 5.1: Graphs of the arguments analyses. Each node represents a possible value of the data-flow. Nodes with a double border represent the initial state. Dashed lines represent the lattice. Solid lines represent the transfer functions, where Read and Write read represent the fact that a register or stack slot has been read or written, IFC represents an indirect function call, and The Call represents the call currently being considered.

```
draof:               raofc:               usaof:
 mov rax,42           mov rax,42           mov rbx,[rsp+8]
 ret                  call lol             ret
                      ret

drvofc:              saofc:               urvofc:
 call lol             call lol             call lol
 mov rax,42           mov rax,[rsp+8]      mov rbx,rax
 ret                  ret                  ret

dsaof:               uraof:               urvof:
 mov [rsp+8],42       mov rbx,rax          mov rax,42
 ret                  ret                  ret
```

Listing 5.3: Example of the relevant code for each argument analysis. The relevant register is always `rax`, or, if the analysis focuses on stack slots the interesting one is at `rsp+8`.

over all paths after the function call, the register is clobbered before being read. See Figure 5.1b.

Dead Stack Arguments Of Function (DSAOF). This analysis is the analogous of DRAOF for stack arguments. The only difference is that its output is not "*if an argument, it's dead*", but simply "*it's dead*". In fact, if a stack slot is ever used directly, it's definitely an argument (despite the fact that it's unused), while a register might just be a scratch register. See Figure 5.1c.

Register Arguments Of Function Call (RAOFC). It detects if a register is an argument of a certain function call. To detect this, RAOFC, checks if no instruction is reading its value between a certain assignment and the considered function call. Under Assumption A2, there's no reason under which such an assignment should be there, unless it's writing an argument for the function call. See Figure 5.1d.

Stack Arguments Of Function Call (SAOFC). It detects if a stack slot is *not* an argument of a specific function call. To detect such a situation, SAOFC, checks if its value is read after the function call before any other instruction writes there. In fact, under Assumption A6, this means that that stack slot is actually not an argument, but simply a local variable, whose value has been preserved across the function call. See Figure 5.1e.

Used Arguments Of Function (URAOF/USAOF). This analysis works both for register arguments (URAOF) and for stack arguments (USAOF). It detects if a certain register or stack slot is read (on at least one path) before being written to. This would mean that the value being read has been passed by the caller, and it's therefore definitely an argument (under Assumption A3). See Figure 5.1f.

Used Return Values Of Function Call (URVOFC). It detects if a register is a return value of a certain function call. To do so, URVOFC, checks if, on at least one path originating in the function call, the value of the register is read before being written to. Under Assumption A3, this definitely means it's a return value. See Figure 5.1g.

Used Return Values Of Function (URVOF). It detects if a register is a return value of a certain function. To do so, URVOF, checks if, on at least one path, the value of a register is not read between an instruction writing to it and the return instruction of the function. Under Assumption A2, this definitely means it's a return value. See Figure 5.1h.

It is important to understand that when we talk about stack slots, we intend memory locations located at a certain offset within the current stack frame (i.e., the `SP0` alias domain). Specifically, we focus on address at a positive offset from the initial value of the stack pointer (i.e., *below* the current function's stack frame). Such addresses are provided by the *stack analysis* due to computations relative to the stack pointer register. Pointer to local variables of the caller passed as an argument do not fall in this category. In fact, they would appear as relative to the caller stack frame (i.e., `SP1`). Therefore, they are not taken into account as potential arguments.

Listing 5.3 presents an example capturing the typical situation where each of the presented analysis is useful. Figure 5.1 presents the lattice and transfer functions graphs, which define the described data-flow analyses (Section 1.5). We developed a tool that, from these graphs, automatically ensures the validity of the lattice, the monotonicity of the transfer functions and generates C++ code implementing the core components of the data-flow analyses. Listing 5.4 reports an example of the code generated for URVOF.

5.2.4 The Final Output

All the presented analyses provide useful information, but, individually, they are not very helpful to the end user. Therefore we need a mechanism to output a final answer about which register or stack slots are arguments of a certain function or function call, and which registers contain a return value for a certain function or function call.

As mentioned in the previous section, in a first pre-processing phase we employ the information provided by the *stack analysis* to identify *explicitly callee-saved registers* and we mark them as *not arguments* and *not return values*, as per Assumption A5. Moreover, we mark all the unused registers as *maybe an argument* and *maybe a return value*, ignoring the results of the analyses, in particular those related to function calls, which would provide information that we cannot trust.

An option to increase the informativeness of our results, at the cost of introducing some approximation is the following: assume that all the registers that have never been explicitly callee-saved in any function are not callee-saved registers (NCS) and assume that all the registers that, in every function, are either unused or *explicitly callee-saved*, are effectively callee-saved registers (CS). Or, in symbols:

```
class UsedReturnValuesOfFunction {
public:
  enum Values { Maybe, Yes };
  enum TransferFunction { Write, Read, IndirectFunctionCall };

public:
  UsedReturnValuesOfFunction() : Value(Maybe) { }
  UsedReturnValuesOfFunction(Values V) : Value(V) { }

  void combine(const UsedReturnValuesOfFunction &RHS) {
    if ((Value == Maybe && Other.Value == Yes) ||
        (Value == Yes && Other.Value == Maybe)) {
      Value = Yes;
    }
  }

  bool
  lowerThanOrEqual(const UsedReturnValuesOfFunction &RHS) const {
    return Value == Other.Value
      || (Value == Maybe && Other.Value == Yes);
  }

  void transfer(TransferFunction T) {
    switch(T) {
    case Write:
      if (Value == Maybe) Value = Yes;
      break;
    case Read:
      if (Value == Yes) Value = Maybe;
      break;
    case IndirectFunctionCall:
      if (Value == Yes) Value = Maybe;
      break;
    }
  }

private:
  Values Value;
};
```

Listing 5.4: C++ code generated for URVOF. The `Values enum` represents the possible values of the data-flow analysis. The `TransferFunction enum` represents the list of different transfer functions available. The default constructor initializes the class value to the initially value, in this case, MAYBE. The `combine` method implements the $\sqcup$ operator, obtained from the graph of the lattice. The `lowerThanOrEqual` method compares two elements of the lattice ($\sqsubseteq$). The `transfer` method applies to the current value the given transfer function.

$$\mathrm{CS} = \bigcap_f \{\mathrm{ECS}_f \cup \mathrm{Unused}_f\} \qquad \mathrm{NCS} = \left\{ r \mid r \notin \bigcup_f \mathrm{ECS}_f \right\}$$

where ECS_f is the set of *explicitly callee-saved registers* in f as detected by the *stack analysis* and Unused_f is the set of registers unused in f.

Such assumptions are particularly reasonable in a *closed world assumption*, i.e., if either all the code is available to the analysis framework, or if the binary is large enough. A similar approach can also be employed to obtain some information from functions calls whose target is unknown.

Once, *explicitly callee-saved* and unused registers have been handled, the following rules are employed to get a final information about stack slots and the remaining registers.

Arguments of a function in registers. A register in a function, in terms of being an argument, can be in one of the following final states:

NoOrDead not an argument or an unused argument.

Dead an unused argument.

Yes a used argument.

Maybe possibly an argument.

Contradiction one or more of our assumptions have been violated.

To produce a final value, for the register r, for the function f we first combine the outputs of DRAOF and URAOF:

		URAOF	
		Maybe/Unknown	Yes
DRAOF	Maybe/Unknown	Maybe	Yes
	NoOrDead	NoOrDead	Contradiction

Then we combine the result with each result of RAOFC for function calls to f:

		RAOFC	
		MaybeMaybe/YesMaybe	StarYes
Old result	NoOrDead	NoOrDead	Dead
	Dead	Dead	Dead
	Contradiction	Contradiction	Contradiction
	Maybe	Maybe	Yes
	Yes	Yes	Yes

Arguments of a function on the stack. A stack slot in a function, in terms of being an argument, can be in one of the following final states:

Dead an unused argument.
Yes a used argument.
No not an argument.
Maybe possibly an argument.
Contradiction one or more of our assumptions have been violated.

To produce a final value, for the stack slot s, for the function f we first combine the outputs of DSAOF and USAOF:

		USAOF	
		Maybe/Unknown	Yes
DSAOF	Maybe/Unknown	Maybe	Yes
	Dead	Dead	Contradiction

Then we combine the result with each result of SAOFC for function calls to f:

		SAOFC	
		Maybe/Unknown	No
Old result	Dead	Dead	Contradiction
	Contradiction	Contradiction	Contradiction
	Maybe	Maybe	No
	Yes	Yes	Contradiction
	No	No	No

Return values of a function. A register in a function, in terms of being a return value, can be in one of the following final states:

NoOrDead not a return value or an unused return value.
Dead an unused return value.
Yes a return value.
Maybe possibly a return value.
YesCandidate from the function point of view, a return value.

To produce a final value, for the register r, for the function f we initialize the result with the values from URVOF (considering its Yes as YesCandidate). Then we combine them with all the instances of the DRVOFC analysis targeting the function f:

		DRVOFC	
		Maybe/Unknown	NoOrDead
Old result	NoOrDead	NoOrDead	NoOrDead
	Maybe	Maybe	NoOrDead
	Yes	Yes	Yes
	Dead	Dead	Dead
	YesCandidate	YesCandidate	Dead

Then we do the same with all the instances of the URVOFC analysis targeting the function f:

		URVOFC	
		Maybe/Unknown	Yes
Old result	NoOrDead	NoOrDead	Yes
	Maybe	Maybe	Yes
	Yes	Yes	Yes
	Dead	Dead	Yes
	YesCandidate	YesCandidate	Yes

Note that the YesCandidate state, when integrated with information from call sites, can either go to Yes, if at least a caller uses it, or to Dead, in case all the callers ignore it.

Return values of a function call. A register, in terms of being a return value of a certain function call, can be in one of the following final states:

NoOrDead not a return value or an unused return value.
Dead an unused return value.
Yes a return value.
Maybe possibly a return value.
Contradiction one or more of our assumptions have been violated.

Note that we need to introduce this distinction, since, while the number of arguments is a property exclusively concerning a function (i.e., the information we provide is good for all the call sites), this is not true for return values. In fact the fact that a return value is used or unused, is a property of the call site, and not of the called function.

We initialize the result with the output of URVOF. Then we combine it with the instance of the DRVOFC analysis for the currently considered call site:

		URVOF	
		Maybe	Yes
DRVOFC	Maybe/Unknown	Maybe	Yes
	NoOrDead	NoOrDead	Dead

Then we do the same with the instances of the URVOFC for the currently considered call site:

		URVOFC	
		Maybe/Unknown	Yes
Old result	NoOrDead	NoOrDead	Contradiction
	Dead	Dead	Contradiction
	Maybe	Maybe	Yes
	Yes	Yes	Yes

Chapter 6

Conclusions

This chapter concludes Part I with some words about current limitations of `rev.ng`, its future development directions and some considerations on related works.

6.1 Limitations and Future Directions

While `rev.ng` can already produce interesting results, being for instance able to correctly handle large pieces of sofware such as GCC and Perl, there's still much work to be done.

Specifically, we need to finalize the support for dynamic libraries described in Section 2.3.2 and assess the quality of the results of our function prototype detection algorithm (Chapter 5), which could not be included in this work due to time constraints. We're currently working on creating standalone LLVM functions using the information about function boundaries provided by the techniques described in Chapter 4 and Chapter 5. This will provide to the LLVM much smaller and cleaner units to optimize, which should lead to sensible speedups.

On a mere engineering level, we intend to extend our platform support by handling Mach-O and PE/COFF image formats (for which we already have a preliminary support). On the longer term we also aim at extending QEMU's user mode to support Windows programs too. This represents the first step towards being able to correctly recompile and run Windows programs.

On a more theoretical side, we're currently in the process of designing a control-flow structuring algorithm, heavily inspired by [167]. The next logical steps will be identification of local variables and type recognition of arguments and variables, which will open the way to our final grand goal: emitting readable and recompilable C code.

6.2 Related Works

Static binary translation. A large part of the work presented along Part I falls in the field of static binary translation, which is a subset of binary translation aiming at decoupling the translation of the binary from its execution. Binary translation has been studied for decades. Early efforts in the 1980s and 1990s focused on porting legacy code, or providing fast emulation platforms, with retargetability soon becoming a key concern [29, 27]. Applications of binary translation beyond the classic legacy code portability problem include binary instrumentation for security enforcement [172], reverse engineering, and de-obfuscation [163].

LLBT [139] is a static binary translator based on LLVM. Similar to `revamb`, this tool employs the LLVM IR to achieve retargetability in the static binary translator. The main difference between LLBT and `revamb` is that we use QEMU's IR to perform the translation from binary to LLVM IR. Thus, `revamb` is inherently easier to maintain than LLBT, and requires much less work to add new source-target architecture pairs, as long as they are supported by QEMU and LLVM respectively. Regarding code discovery, LLBT focuses on ARM architectures and implements an ad-hoc mechanism to recover common patterns, limiting the generality of the approach. Their technique is very effective, as only 25% spurious regions are translated, but specific to the ARM and Thumb ISAs.

UROBOROS [164] is a tool that focuses on producing disassembled code which can be reassembled without manual effort. It currently supports the disassembly of ELF binaries for the x86 and x64 architectures. The key challenge tackled by UROBOROS is to make the disassembled code relocatable by means of *symbolization*. Symbolization basically consists in trying to identify all the references to code (both in code and data) and make them relocatable. To do this, a set of heuristics is employed, with serious limitations in terms of accuracy and introducing the possibility of severe corruptions in the program's behavior. On the other hand, our approach is much more principled and conservative (see the discussion about overtranslation in Section 2.2.2). Moreover, unlike `rev.ng`, UROBOROS is not meant to support recompilation to a different architecture.

CodeSurfer/x86 [10] is a tool based on IDA Pro [72] and CodeSurfer [66], which implements *value-set analysis* (VSA), a form of data-flow analysis which tracks the contents of memory addresses, providing an over-approximation of the set of values that can be held in a memory location or register at each program point. A key difference with our tool is that we support multiple architectures instead of just x86, and that, among other things, OSRA (Section 3.3) can provide more fine-grained and precise information about destinations of indirect jumps due to `switch` statements.

Cifuentes and Emmerik [28] proposed a slicing analysis to identify jump targets from `switch` constructs, which is effective and portable, but does not deal with indirect calls, which causes under-translation in several cases. These effects are countered in their binary translation framework via an interpreter, which makes it necessary to use a dynamic rather than static binary translation technique.

Similar issues have been studied in Jakstab [85, 84], an abstract interpretation-based, integrated disassembly and static analysis framework for designing analyses, which is however limited to x86-64.

Function boundary identification. Traditional techniques used to identify function entry points employ manually crafted patterns and then use recursive disassembly to identify the set of bytes belonging to a function body. Such techniques are adopted in current tools, both commercial and research-oriented, such as IDA Pro [72], Dyninst [99, 69], as well as in other disassemblers [162], and angr [141, 142].

angr [141, 142, 146] adopts an approach similar to ours, since it employs VEX, Valgrind's IR, to perform their analysis. However, VEX is only available for a subset of the architecture handled by QEMU. Most importantly, the largest part of their effort for accurate recovery of CFG and function boundaries relies on symbolic execution, which, despite providing extremely accurate results, severely hinders the scalability of the approach to the point that for larger binaries angr has often to fall back to a set of heuristics.

Rosenblum et al. [130] employed machine learning to address function boundary identification, overcoming variation in the function start due to compiler-related effects such as optimization or scheduling. Basically, they proposed to automatically generate the set of function start patterns from a large corpus of binaries, instead of crafting it manually.

ByteWeight [11] refines this idea, leveraging machine learning classification to label each byte of a program as a function start or not. It employs weighted prefix trees of function start sequences in place of a pattern collection, followed by static analysis (recursive disassembly combined with VSA [10]) to detect the remaining bytes of the function. Shin et al. [140] aim at improving precision and speed of recovery over ByteWeight employing recurrent neural networks.

ByteWeight is based on BAP [20], an OCaml binary analysis platform for ARM and x86 platforms, which also employs and intermediate representation for its analyses. BAP is a rewrite of BitBlaze [144], a tool that employs the GNU disassembler and VEX (Valgrind's IR) to lift x86 instructions to a custom intermediate representation known as Vine.

For all of the above mentioned machine learning-based approaches, the main goal is to reconstruct a set of probable start patterns. This often leads to mistakes in case of instructions that might look like a function prologue but are not, such as calls to `alloca` or the construction of Variable Length Arrays. Our technique is fundamentally different, in that it relies on code pointers to identify function starting points. We leverage data-flow analyses that can provide more fine-grained, principled and precise information about the tracked values with respect to value set analysis, as shown in Chapter 3.

Andriesse et al. [8] present an interesting comparison of the quality of the function boundary detection techniques published in top conferences in the latest years, showing varying quality results.

Andriesse et al. [7] created a structural control flow graph analysis which is part of the tool named Nucleus. Their approach is very similar to the one we presented in Chapter 4 and published in [51], which predates their work.

Function prototype identification. Very little work has been published on the detection of function arguments and return values in an ABI-independent fashion.

As a part of TypeArmor van der Veen and Göktas et al. [159] proposed a custom inter-procedural liveness analysis for x64 binaries to determine register arguments of function in a conservative way. Their work, while very preliminary and focused on registers only, inspired a large part of the analyses presented in Chapter 5.

Similar concepts were also briefly introduced in [161], where the function boundary and prototype detection of the Retargetable Decompiler [160, 88] were presented.

Part II

Compiler-aided Binary Hardening

Chapter 7

HexVASAN: a Variadic Function Sanitizer

Programming languages such as C and C++ support variadic functions, i.e., functions that accept a variable number of arguments (e.g., printf). While variadic functions are flexible, they are inherently not type-safe. In fact, the semantics and parameters of variadic functions are defined implicitly by their implementation. It is left to the programmer to ensure that the caller and callee follow this implicit specification, without the help of a static type checker. An adversary can take advantage of a mismatch between the argument types used by the caller of a variadic function and the types expected by the callee to violate the language semantics and to tamper with memory. Format string attacks are the most popular example of such a mismatch.

Indirect function calls can be exploited by an adversary to divert execution through illegal paths. Mechanisms such as CFI can restrict call targets according to the function prototype which, for variadic functions, doesn't include variadic arguments. Therefore, current CFI implementations are mainly limited to non-variadic functions and fail to address this potential attack vector. Defending against such an attack requires a stateful dynamic check.

In this chapter, we present HexVASAN, a compiler based sanitizer to effectively type-check and thus prevent any attack via variadic functions. The key idea is to record metadata about the passed arguments on the call site side and verify the number and type of arguments used by the callee are compatible. Our evaluation shows that HexVASAN is effective and practically deployable with a negligible overhead (0.45%).

This chapter is in large parts extracted from [16].

7.1 Introduction

C and C++ are popular languages in systems programming. This is mainly due to their low overhead abstractions and high degree of control left to the developer. However, these languages guarantee neither type nor memory safety, and bugs may lead to memory corruption. Memory corruption attacks allow adversaries to take control of vulnerable applications or to extract sensitive information.

Modern operating systems and compilers implement several defense mechanisms to combat memory corruption attacks. The most prominent defenses are Address Space Layout Randomization (ASLR) [115], stack canaries [32], and Data Execution Prevention (DEP) [116]. While these defenses raise the bar against exploitation, sophisticated attacks are still feasible. In fact, even the combination of these defenses can be circumvented through information leakage and code-reuse attacks.

Stronger defense mechanisms such as Control Flow Integrity (CFI) [5], protect applications by restricting their control flow to a predetermined control-flow graph (CFG). While CFI allows the adversary to corrupt non-control data, it will terminate the process whenever the control-flow deviates from the predetermined CFG. The strength of any CFI scheme hinges on its ability to statically create a precise CFG for indirect control-flow edges (e.g., calls through function pointers in C or virtual calls in C++). Due to ambiguity and imprecision in the analysis, CFI restricts adversaries to an over-approximation of the possible targets of individual indirect call sites.

We present a new attack against widely deployed mitigations through a frequently used feature in C/C++ that has so far been overlooked: variadic functions. Variadic functions (such as `printf`) accept a varying number of arguments with varying argument types. To implement variadic functions, the programmer implicitly encodes the argument list in the semantics of the function and has to make sure the caller and callee adhere to this implicit contract. In `printf`, the expected number of arguments and their types are encoded implicitly in the format string, the first argument to the function. Another frequently used scheme iterates through parameters until a condition is reached (e.g., a parameter is `NULL`). Listing 7.1 shows an example of a variadic function. If an adversary can violate the implicit contract between caller and callee, an attack may be possible.

In the general case, it is impossible to enumerate the arguments of a variadic function through static analysis techniques. In fact, their number and types are intrinsic in how the function is defined. This limitation enables (or facilitates) two attack vectors against variadic functions. First, attackers can hijack indirect calls and thereby call variadic functions over control-flow edges that are never taken during any legitimate execution of the program. Variadic functions that are called in this way may interpret the variadic arguments differently than the function for which these arguments were intended, and thus violate the implicit caller-callee contract. CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type signature of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions.

A second attack vector involves overwriting a variadic function's arguments directly. Such attacks do not violate the intended control flow of a program and thus bypass all of the widely deployed defense mechanisms. Format string attacks are a prime example of such attacks. If an adversary can control the format string passed to, e.g., `printf`, she can control how all of the following parameters are interpreted, and can potentially leak information from the stack, or read/write to arbitrary memory locations.

We analyzed popular software packages, such as Firefox, Chromium, Apache, CPython, nginx, OpenSSL, Wireshark, the SPEC CPU2006 benchmarks, and the FreeBSD base system, and found that variadic functions are ubiquitous. The underlying problem that enables attacks on variadic functions is the lack of type checking. Variadic functions generally do not (and cannot) verify that the number and type of arguments they expect matches the number and type of arguments passed by the caller.

We present HexVASAN, a compiler sanitizer that tackles this problem by instrumenting the generated code to perform the necessary checks at run-time. Each argument that is retrieved in a variadic function is type checked, enforcing a strict contract between caller and callee so that (i) a maximum of the passed arguments can be retrieved and (ii) the type of the arguments used at the callee are compatible with the types passed by the caller.

We have implemented HexVASAN on top of the LLVM compiler framework, instrumenting the compiled code to record the types of each argument of a variadic function at the call site and to check the types whenever they are retrieved. Our prototype implementation is light-weight, resulting in a negligible (0.45%) overhead for SPEC CPU2006. Our approach is general as we show by recompiling the FreeBSD base system and effective as shown through several exploit case studies (e.g., a format string vulnerability in sudo).

7.2 Background

Variadic functions are a popular feature in C/C++ programs. In this section we introduce details about their use and implementation on current systems, the attack surface they provide, and how adversaries can abuse them.

7.2.1 Variadic Functions

Variadic functions (such as the printf function in the C standard library) are used in C to maximize the flexibility in the interface of a function, allowing it to accept a number of arguments unknown at compile-time. These functions accept a variable number of arguments, which do not necessarily have fixed types. An example of a variadic function is shown in Listing 7.1. The function add accepts one mandatory argument (start) and a varying number of additional arguments, which are marked by the ellipsis (...) in the function definition.

The C standard defines several macros that portable programs may use to access variadic arguments [92]. stdarg.h, the header that declares these macros, defines an opaque type, va_list, which stores all information required to retrieve and iterate through variadic arguments. In our example, the variable list of type va_list is initialized using the va_start macro. The va_arg macro retrieves the next variadic argument from the va_list, updating va_list to point to the next argument as a side effect. Note that, although the programmer must specify the expected type of the variadic argument in the call to va_arg, the C standard does

```c
#include <stdio.h>
#include <stdarg.h>

int add(int start, ...) {
  int next, total = start;
  va_list list;
  va_start(list, start);
  do {
    next = va_arg(list, int);
    total += next;
  } while (next != 0);
  va_end(list);
  return total;
}

int main(int argc, const char *argv[]) {
  printf("%d\n", add(5, 1, 2, 0));
  return 0;
}
```

Listing 7.1: Example of a variadic function in C. The function `add` takes a non-variadic argument `start` (to initialize an accumulator variable) and a series of variadic `int` arguments that are added until the terminator value `0` is met. The final value is then returned.

not require the compiler to verify that the retrieved variable is indeed of that type. `va_list` variables must be released using a call to the `va_end` macro so that all of the resources assigned to the list are deallocated.

`printf` is an example of a more complex variadic function which takes a format string as its first argument. This format string implicitly encodes information about the number of arguments and their type. Implementations of `printf` scan through this format string several times to identify all format arguments and to recover the necessary space in the output string for the specified types and formats. Interestingly, arguments do not have to be encoded sequentially but format strings allow out-of-order access to arbitrary arguments. This flexibility is often abused in format string attacks to access arbitrary stack locations.

7.2.2 Variadic Functions ABI

The C standard does not define the calling convention for variadic functions, nor the exact representation of the `va_list` structure. This information is instead part of the ABI of the target platform.

x86-64 ABI. The AMD64 System V ABI [96], which is implemented by x86-64 GNU/Linux platforms, dictates that the caller of a variadic function must adhere to the normal calling conventions when passing arguments. Specifically, the first six non-floating point arguments and the first eight floating point arguments are passed through CPU registers. The remaining arguments, if any, are passed on the stack. If a variadic function accepts five mandatory arguments and a variable number of variadic arguments, then all but one of these variadic arguments will be passed on the stack. The variadic function itself moves the arguments into a va_list variable using the va_start macro. The va_list type is defined as follows:

```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
} va_list[1];
```

va_start allocates on the stack a reg_save_area to store copies of all variadic arguments that were passed in registers. va_start initializes the overflow_arg_area field to point to the first variadic argument that was passed on the stack. The gp_offset and fp_offset fields are the offsets into the reg_save_area. They represent the first unused variadic argument that was passed in a general purpose register or floating point register respectively.

The va_arg macro retrieves the first unused variadic argument from either the reg_save_area or the overflow_arg_area, and either it increases the gp_offset/fp_offset field or moves the overflow_arg_area pointer forward, to point to the next variadic argument.

Other architectures. Other architectures may implement variadic functions differently. On 32-bit x86, for example, all variadic arguments must be passed on the stack (pushed right to left), following the cdecl calling convention used on GNU/Linux. The variadic function itself retrieves the first unused variadic argument directly from the stack. This simplifies the implementation of the va_start, va_arg, and va_end macros, but it generally makes it easier for adversaries to overwrite the variadic arguments.

7.2.3 Variadic Attack Surface

When calling a variadic function, the compiler statically type checks all non-variadic arguments but does not enforce any restriction on the type or number of variadic arguments. The programmer must follow the implicit contract between caller and callee that is only present in the code but never enforced explicitly. Due to this high flexibility, the compiler cannot check arguments statically. This lack of safety can lead to bugs where an adversary achieves control over the callee by modifying the arguments, thereby influencing the interpretation of the passed variadic arguments.

Modifying the argument or arguments that control the interpretation of variadic arguments allows an adversary to change the behavior of the variadic function, causing the callee to access additional or fewer arguments than specified and to change the interpretation of their types.

An adversary can influence variadic functions in several ways. First, if the programmer forgot to validate the input, the adversary may directly control the arguments to the variadic function that controls the interpretation of arguments.

Second, the adversary may use an arbitrary memory corruption elsewhere in the program to influence the argument of a variadic function.

Variadic functions can be called statically or dynamically. Direct calls would, in theory, allow some static checking. Indirect calls (e.g., through a function pointer), where the target of the variadic function is not known, do not allow any static checking. Therefore, variadic functions can only be protected through some form of runtime checker that considers the constraints of the call site and enforces them at the callee side.

7.2.4 Format String Exploits

Format string exploits are a perfect example of corrupted variadic functions. An adversary that gains control over the format string used in `printf` can abuse the `printf` function to leak arbitrary data on the stack or even resort to arbitrary memory corruption (if the pointer to the target location is on the stack). For example, a format string vulnerability in the smbclient utility (CVE-2009-1886) [106] allows an attacker to gain control over the Samba file system by treating a filename as format string. Also, in PHP before 7.0.1, an error handling function allows an attacker to execute arbitrary code by using format string specifiers as class name (CVE-2015-8617) [1].

Information leaks are simple: an adversary changes the format string to print the desired information that resides somewhere higher up on the stack by employing the desired format string specifiers. For arbitrary memory modification, an adversary must have the target address encoded somewhere on the stack and then reference the target through the `%n` modifier, writing the number of already written bytes to that memory location.

The GNU C standard library (*glibc*) enforces some protection against format string attacks by checking if a format string is in a writable memory area [83]. For format strings, the *glibc* `printf` implementation opens `/proc/self/maps` and scans for the memory area of the format string to verify correct permissions. Moreover, a check is performed to ensure that all arguments are consumed, so that no out-of-context stack slots can be used in the format string exploit. These defenses stop some attacks but do not mitigate the underlying problem that an adversary can gain control over the format string. Note that this heavyweight check is only used if the format string argument *may* point to a writable memory area at compile time. An attacker may use memory corruption to redirect the format string pointer to an attacker-controlled area and fall back to a regular format string exploit.

7.3 Threat Model

Programs frequently use variadic functions, either in the program itself or as part of a shared library (e.g., `printf` in the C standard library). We assume that the program contains an arbitrary memory corruption, allowing the adversary to modify the arguments to a variadic function and/or the target of an indirect function call, targeting a variadic function.

Our target system deploys existing defense mechanisms like DEP, ASLR, and a strong implementation of CFI, protecting the program against code injection and control-flow hijacking. We assume that the adversary cannot modify the metadata of our runtime monitor. Protecting metadata is an orthogonal engineering problem and can be solved through, e.g., masking (`and`-ing every memory access), segmentation (for x86-32), protecting the memory region [23], or randomizing the location of sensitive data.

7.4 Design

`HexVASAN` monitors calls to variadic functions and checks for type violations. Since the semantics of how arguments should be interpreted by the function are intrinsic in the logic of the function itself, it is, in general, impossible to determine the number and type of arguments a certain variadic function accepts. For this reason, `HexVASAN` instruments the code generated by the compiler so that a check is performed at runtime. This check ensures that the arguments consumed by the variadic function match those passed by the caller.

The high level idea is the following: `HexVASAN` records metadata about the supplied argument types at the call site and verifies that the extracted arguments match in the callee. The number of arguments and their types is always known at the call site and can be encoded efficiently. In the callee this information can then be used to verify individual arguments when they are accessed. To implement such a sanitizer, we must design a metadata store, a pass that instruments call sites, a pass that instruments callers, and a runtime library that manages the metadata store and performs the run-time type verification. Our runtime library aborts the program whenever a mismatch is detected and generates detailed information about the call site and the mismatched arguments.

7.4.1 Analysis and Instrumentation

We designed `HexVASAN` as a compiler pass to be run in the compilation pipeline right after the C/C++ front-end. The instrumentation collects a set of statically available information about the call sites, encodes it in the LLVM module, and injects calls to our runtime to perform checks during program execution.

Figure 7.1 provides an overview of the compilation pipeline when `HexVASAN` is enabled. Source files are first parsed by the C/C++ frontend which generates the intermediate representation on which our instrumentation runs. The normal compilation then proceeds, generating instrumented object files. These object files, along with the `HexVASAN` runtime library, are then passed to the linker, which creates the instrumented program binary.

7.4.2 Runtime Support

The `HexVASAN` runtime augments every `va_list` in the original program with the type information generated by our instrumentation pass, and uses this type information to perform run-time type checking on any variadic argument accessed

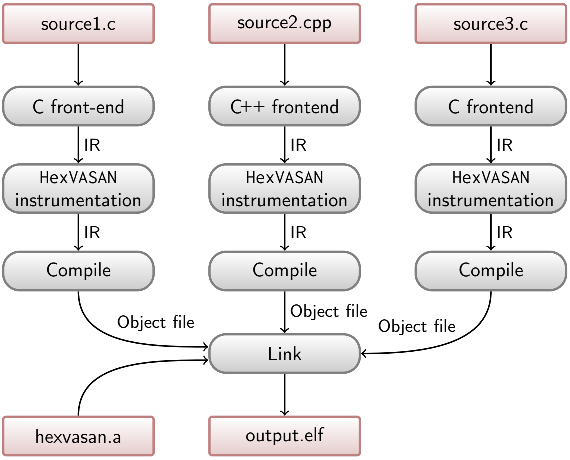


Figure 7.1: Overview of the HexVASAN compilation pipeline. The HexVASAN instrumentation runs right after the C/C++ frontend, while its runtime library, hexvasan.a, is merged into the final executable at link time.

through va_arg. By managing the type information in a metadata store, and by maintaining a mapping between va_lists and their associated type information, HexVASAN remains fully compatible with the platform ABI.

The HexVASAN runtime manages the type information in two data structures. The core data structure, called the *variadic list map* (VLM), associates va_list structures with the type information produced by our instrumentation, and with a counter to track the index of the last argument that was read from the list. A second data structure, the *variadic call stack* (VCS), allows callers of variadic functions to store type information of variadic arguments until the callee initializes the va_list.

Each variadic call site is instrumented with a call to pre_call, that prepares the information about the call site (a *variadic call site descriptor* or VCSD), and a call to post_call, that cleans it up. For each variadic function, the va_start calls are instrumented with list_init, while va_copy, whose purpose is to clone a va_list, is instrumented through list_copy. The two run-time functions will allocate the necessary data structures to validate individual arguments. Calls to va_end are instrumented through list_end to free up the corresponding data structures.

Algorithm 7.1 summarizes the two phases of our analysis and instrumentation pass. The first phase identifies all the calls to variadic functions (both direct and indirect). Note that identifying indirect calls to variadic functions is straight-forward in a compiler framework since, even if the target function is not statically known, its type is. Then, all the parameters passed by that specific call site are inspected and recorded, along with their type in a dedicated VCSD which is stored in read-only global data. At this point, a call to pre_call is injected before the variadic function call (with the newly created VCSD as a parameter) and, symmetrically, a call to post_call is inserted after the call site.

The second phase identifies all calls to va_start and va_copy, and consequently, the va_list variables in the program. Uses of each va_list variable are inspected in an architecture-specific way. Once all uses are identified, we inject a call to check_arg before dereferencing the argument (which always resides in memory).

7.4.3 Challenges and Discussion

When designing a variadic function call sanitizer, several issues have to be considered. We highlight details about the key challenges we encountered.

Multiple va_lists. Functions are allowed to create multiple va_lists to access the same variadic arguments, either through va_start or va_copy operations. HexVASAN handles this by storing a VLM entry for each individual va_list.

Passing va_lists as function arguments. Variadic functions are allowed to pass the va_lists they create as arguments to non-variadic functions. This allows non-variadic functions to access variadic arguments of functions higher in the call stack. Our design takes this into account by maintaining a list map (VLM) and by instrumenting all va_arg operations, regardless of whether or not they are in a

```
input: a module m
/* Phase 1                                                         */
foreach function f in module m do
    foreach variadic call c with n arguments in f do
        vcsd.count ← n;
        foreach argument a of type t do
            vcsd.args.push(t);
        end
        emit call to pre_call(vcsd) before c;
        emit call to post_call() after c;
    end
end
/* Phase 2                                                         */
foreach function f in module m do
    foreach call c to va_start(list) do
        emit call to list_init(&list) after c;
    end
    foreach call c to va_copy(dst, src) do
        emit call to list_copy(&dst, &src) after c;
    end
    foreach call c to va_end(list) do
        emit call to list_free(&list) after c;
    end
    foreach call c to va_arg(list, type) do
        emit call to check_arg(&list, type) before c;
    end
end
```

Algorithm 7.1: The instrumentation process.

variadic function.

Multi-threading support. Multiple threads are supported by storing our per-thread runtime state in a thread-local variable as supported on major operating systems.

Metadata format. We use a constant data structure per variadic call site, the VCSD, to hold the number of arguments and a pointer to an array of integers identifying their type. The `check_arg` function therefore only performs two memory accesses, the first to load the number of arguments and the second for the type of the argument currently being checked.

To uniquely identify the data types with an integer, we decided to build a hashing function (described in Algorithm 7.2) using a set of fixed identifiers for primitive data types and hashing them in different ways depending on how they are aggregated (pointers, `union`, or `struct`). The last hash acts as a terminator marker for aggregate types, which allows us to, e.g., distinguish between:

```
struct One {
  struct {
    int a;
  } x;
  int b, c;
};
```

```
struct Two {
  struct {
    int a, b;
  } x;
  int c;
};
```

Note that an (unlikely) hash collision only results in two different types being accepted as equal. Such a hashing mechanism has the advantage of being deterministic across compilation units, removing the need for keeping a global map of type-unique id pairs. Due to the information loss during the translation from C/C++ to LLVM IR, our type system does not distinguish between signed and unsigned types. The required metadata is static and immutable and we mark it as read-only, protecting it from modification. However, the VCS still needs to be protected through other mechanisms.

Handling floating point arguments. In x86-64 ABI, floating point and non-floating point arguments are handled differently. In case of floating point arguments, the first eight arguments are passed in the floating point registers whereas in case of non-floating point the first six are passed in general-purpose registers. HexVASAN handles both argument types.

Support for aggregate data types. According to AMD64 System V ABI, the caller unpacks the fields of the aggregate data types (`struct`s and `union`s) if the arguments fit into registers. This makes it hard to distinguish between composite types and regular types – if unpacked they are indistinguishable on the callee side from arguments of these types. HexVASAN supports aggregate data types even if the caller unpacks them.

Attacks preserving number and type of arguments. Our mechanism prevents attacks that change the number of arguments or the types of individual arguments. Format string attacks that only change one modifier can therefore be detected through the type mismatch even if the total number of arguments remains unchanged.

```
input  : a type $t$ and an initial hash value $h$
output: the final hash value $h$
$h = $ hash($h$, typeID($t$));
switch typeID($t$) do
    case AggregateType do
        /* union, struct and pointer                    */
        foreach $c$ in componentTypes($t$) do
            $h = $ hashType($c$, $h$);
        end
    case FunctionType do
        $h = $ hashType(returnType($t$), $h$);
        foreach $a$ in argTypes($t$) do
            $h = $ hashType($a$, $h$);
        end
    end
end
$h = $ hash($h$, typeID($t$));
return h
```

Algorithm 7.2: Algorithm describing the type hashing function *hashType*. *typeID* returns an unique identifier for each basic type (e.g., 32-bit integer, `double`), type of aggregate type (e.g., `struct`, `union`...) and functions. *hash* is a simple hashing function combining two integers. *componentTypes* returns the components of an aggregate type, *returnType* the return type of a function prototype and *argTypes* the type of its arguments.

Non-variadic calls to variadic functions. Consider the following code snippet:

```
typedef void (*non_variadic)(int, int);

void variadic(int, ...) { /* ... */ }

int main() {
  non_variadic function_ptr = variadic;
  function_ptr(1, 2);
}
```

In this case, the function call in `main` to `function_ptr` appears to the compiler as a non-variadic function call, since the type of the function pointer is not variadic. Therefore, our pass will not instrument the call site, leading to potential errors.

To handle such (rare) situations appropriately, we would have to instrument all non-variadic call sites too, leading to an unjustified overhead. Moreover, the code above represents *undefined behavior* in C [81, 6.3.2.3p8] and C++ [80, 5.2.10p6], and might not work on certain architectures where the calling convention for variadic and non-variadic function calls are not compatible. The GNU C compiler emits a warning when a function pointer is cast to a different type, therefore we require the developer to correct the code before applying `HexVASAN`.

Central management of the global state. To allow the `HexVASAN` runtime to be linked into the base system libraries, such as the C standard library, we made it a static library. Turning the runtime into a shared library is possible, but would prohibit its use during the early process initialization – until the dynamic linker has processed all of the necessary relocations. Our runtime therefore either needs to be added solely to the C standard library (so that it is initialized early in the startup process) or the runtime library must carefully use weak symbols to ensure that each symbol is only defined once if multiple libraries are compiled with our countermeasure.

C++ exceptions and longjmp. If an exception is raised while executing a variadic function (or one of its callees), the variadic function may not get a chance to clean up the metadata for any va_lists it has initialized, nor may the caller of this variadic function get the chance to clean up the type information it has pushed onto the VCS. Other functions manipulating the thread's stack directly, such as longjmp, present similar issues.

C++ exceptions can be handled by modifying the LLVM C++ frontend (i.e., clang) to inject an object with a lifetime spanning from immediately before a variadic function call to immediately after. Such an object would call pre_call in its constructor and post_call in the destructor, leveraging the exception handling mechanism to make HexVASAN exception-safe. Functions like longjmp can be instrumented to purge the portions of HexVASAN's data structures that correspond to the discarded stack area. We did not observe any such calls in practice and leave the implementation of handling exceptions and longjmp across variadic functions as future work.

7.5 Implementation

We implemented HexVASAN as a sanitizer for the LLVM compiler framework [89], version 3.9.1. HexVASAN has been publicly release as Free Software [15]. To enable the sanitizer the -fsanitize=vasan switch must be provided to the C/C++ frontend (clang). No annotations or other source code changes are required for HexVASAN. Our sanitizer does not require visibility of whole source code (see Section 7.4.3), but works on individual compilation units. Therefore link-time optimization (LTO) is not required and thus fits readily into existing build systems. In addition, HexVASAN also supports signal handlers.

HexVASAN consists of two components: a static instrumentation pass and a runtime library. The static instrumentation pass works on LLVM IR, adding the necessary instrumentation code to all variadic functions and their callees. The support library is statically linked to the program and, at run-time, checks the number and type of variadic arguments as they are used by the program. In the following we describe the two components in detail.

Static instrumentation. The implementation of the static instrumentation pass follows the description in Section 7.4. We first iterate through all functions, looking for CallInst instructions targeting a variadic function (either directly or indirectly), then we inspect them and create for each one of them a read-only GlobalVariable of type vcsd_t. As shown in Listing 7.2, vcsd_t is composed by

```
struct vcsd_t { unsigned count; type_t *args; };

thread_local stack<vcsd_t *> vcs;
thread_local map<va_list *, pair<vcsd_t *, unsigned>> vlm;

void pre_call(vcsd_t *arguments) { vcs.push_back(arguments); }
void post_call() { vcs.pop_back(); }
void list_init(va_list *list_ptr) {
  vlm[list_ptr] = { vcs.top(), 0 };
}
void list_free(va_list *list_ptr) { vlm.erase(list_ptr); }

void check_arg(va_list *list_ptr, type_t type) {
  pair<vcsd_t *, unsigned> &args = vlm[list_ptr];
  unsigned index = args.second++;
  assert(index < args.first->count);
  assert(args.first->args[index] == type);
}

int add(int start, ...) {
  /* ... */
  va_start(list, start);
  list_init(&list);
  do {
    check_arg(&list, typeid(int));
    total += va_arg(list, int);
  } while (next != 0);
  va_end(list);
  list_free(&list);
  /* ... */
}

const vcsd_t main_add_vcsd = {
  .count = 3,
  .args = { typeid(int), typeid(int), typeid(int) }
};

int main(int argc, const char *argv[]) {
  /* ... */
  pre_call(&main_add_vcsd);
  int result = add(5, 1, 2, 0);
  post_call();
  printf("%d\n", result);
  /* ... */
}
```

Listing 7.2: Simplified C++ representation of the instrumented code for Listing 7.1.

an unsigned integer representing the number of arguments of the considered call site and a pointer to an array (another GlobalVariable) with an integer element for each argument of type_t. type_t is an integer uniquely identifying a data type obtained using the *hashType* function presented in Algorithm 7.2. At this point a call to pre_call is injected before the call site, with the newly create VCSD as a parameter, and a call to post_call is injected after the call site.

During the second phase, we first identify all va_start, va_copy, and va_end operations in the program. In the IR code, these operations appear as calls to the LLVM intrinsics llvm.va_start, llvm.va_copy, and llvm.va_end. We instrument the operations with calls to our runtime's list_init, list_copy, and list_free functions respectively. We then proceed to identify va_arg operations. Although the LLVM IR has a dedicated va_arg instruction, it is not used on any of the platforms we tested. The va_list is instead accessed directly. Our identification of va_arg is therefore platform-specific. On x86-64, our primary target, we identify va_arg by recognizing accesses to the gp_offset and fp_offset fields in the x86-64 version of the va_list structure (see Section 7.2.2). The fp_offset field is accessed whenever the program attempts to retrieve a floating point argument from the list. The gp_offset field is accessed to retrieve any other types of variadic arguments. We insert a call to our runtime's check_arg function before the instruction that accesses this field.

Listing 7.2 shows (in simplified C) how the code in Listing 7.1 would be instrumented by our sanitizer.

Dynamic variadic type checking. The entire runtime is implemented in plain C code, as this allows it to be linked into the standard C library without introducing a dependency to the standard C++ library. The VCS is implemented as a thread-local stack, and the VLM as a thread-local hash map. The pre_call and post_call functions push and pop type information onto and from the VCS. The list_init function inserts a new entry into the VLM, using the top element on the stack as the entry's type information and initializing the counter for consumed arguments to 0.

check_arg looks up the type information for the va_list being accessed in the VLM and checks if the requested argument exists (based on the counter of consumed arguments), and if its type matches the one provided by the caller. If either of these checks fails, execution is aborted, and the runtime will generate an error message such as the one shown in Listing 7.3. As a consequence, the pointer to the argument is never read or written, since the pointer to it is never dereferenced.

7.6 Evaluation

In this section we present a case study on variadic function based attacks against state-of-the-art CFI implementations. Next, we evaluate the effectiveness of HexVASAN as an exploit mitigation technique. Then, we evaluate the overhead introduced by our HexVASAN prototype implementation on the SPEC CPU2006 integer (CINT2006) benchmarks. We also evaluate how widespread the usage of variadic functions is in SPEC CPU2006 and in Firefox 51.0.1, Chromium 58.0.3007.0, Apache 2.4.23,

```
Error: Type Mismatch
Index is 1
Callee Type : 43 (32-bit Integer)
Caller Type : 15 (Pointer)
Backtrace:
[0] 0x4019ff <__vasan_backtrace+0x1f> at test
[1] 0x401837 <__vasan_check_arg+0x187> at test
[2] 0x8011b3afa <__vfprintf+0x20fa> at libc.so.7
[3] 0x8011b1816 <vfprintf_l+0x86> at libc.so.7
[4] 0x801200e50 <printf+0xc0> at libc.so.7
[5] 0x4024ae <main+0x3e> at test
[6] 0x4012ff <_start+0x17f> at test
```

Listing 7.3: Error message reported by HexVASAN.

CPython 3.7.0, nginx 1.11.5, OpenSSL 1.1.1, Wireshark 2.2.1, and the FreeBSD 11.0 base system.

Note that, along with testing the aforementioned software, we also developed an internal set of regression tests. Our regression tests allow us to verify that our sanitizer correctly catches problematic variadic function calls, and does not raise false alarms for benign calls. The test suite explores corner cases, including trying to access arguments that have not been passed and trying to access them using a type different from the one used at the call site.

7.6.1 Case Study: CFI Effectiveness

One of the attack scenarios we envision is that an attacker controls the target of an indirect call site. If the intended target of the call site was a variadic function, the attacker could illegally call a different variadic function that expects different variadic arguments than the intended target (yet shares the types for all non-variadic arguments). If the intended target of the call site was a non-variadic function, the attacker could call a variadic function that interprets some of the intended target's arguments as variadic arguments.

All existing CFI mechanisms allow such attacks to some extent. The most precise CFI mechanisms, which rely on function prototypes to classify target sets (e.g., LLVM-CFI, piCFI, or VTV) will allow all targets with the same prototype, possibly restricting to the subset of functions whose addresses are taken in the program. This is problematic for variadic functions, as only non-variadic types are known statically. For example, if a function of type `int (*)(int, ...)` is expected to be called from an indirect call site, then precise CFI schemes allow calls to all other variadic functions of that type, even if those other functions expect different types for the variadic arguments.

A second way to attack variadic functions is to overwrite their arguments directly. This happens, for example, in format string attacks, where an attacker can overwrite the format string to cause misinterpretation of the variadic arguments. HexVASAN detects both of these attacks when the callee attempts to retrieve the

Intended	Actual target		LLVM-CFI	pi-CFI	CCFI	VTV	CFG	HexVASAN
target	Prototype	A.T.?						
Variadic	Same	Yes	✗	✗	✗	✗	✗	✓
		No	✗	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
Non-variadic	Same	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✗	✗	✗	✓
	Different	Yes	✓	✓	✗	✗	✗	✓
		No	✓	✓	✓	✗	✗	✓
Original	Overwritten args		✗	✗	✗	✗	✗	✓

Table 7.1: Detection coverage for several types of illegal calls to variadic functions. ✓ indicates detection, ✗ indicates non-detection. "A.T." stands for *address taken*.

variadic arguments using the `va_arg` macro described in Section 7.2.1. Checking and enforcing the correct types for variadic functions is only possible at runtime and any sanitizer must resort to run-time checks to do so. CFI mechanisms must therefore be extended with a HexVASAN-like mechanism to detect violations. To show that our tool can complement CFI, we create test programs containing several variadic functions and one non-variadic function. The definitions of these functions are shown below.

```c
int sum_ints(int n, ...);
int avg_longs(int n, ...);
int avg_doubles(int n, ...);
void print_longs(int n, ...);
void print_doubles(int n, ...);
int square(int n);
```

This program contains one indirect call site from which only the `sum_ints` function can be called legally, and one indirect call site from which only the `square` function can be legally called. We also introduce a memory corruption vulnerability which allows us to override the target of both indirect calls.

We constructed the program such that `sum_ints`, `avg_longs`, `print_longs`, and `square` are all address-taken functions. The `avg_doubles` and `print_doubles` functions are not address-taken.

Functions `avg_longs`, `avg_doubles`, `print_longs`, and `print_doubles` all expect different variadic argument types than function `sum_ints`. Functions `sum_ints`, `avg_longs`, avg_doubles, and square do, however, all have the same non-variadic prototype (int (*)(int)).

We compiled six versions of the test program, instrumenting them with, respectively, HexVASAN, LLVM 3.9 Forward-Edge CFI [151], Per-Input CFI [110], CCFI [95], GCC 6.2's VTV [151] and Visual C++ Control Flow Guard [100]. In each version, we first built an attack involving a variadic function, by overriding the indirect call sites with a call to each of the variadic functions described above. We then also tested overwriting the arguments of the sum_ints function, without overwriting the indirect call target. Table 7.1 shows the detection results.

LLVM Forward-Edge CFI allows calls to avg_longs and avg_doubles from the sum_ints indirect call site because these functions have the same static type signature as the intended call target. This implementation of CFI does not allow calls to variadic functions from non-variadic call sites, however.

CCFI only detects calls to print_doubles, a function that is not address-taken and has a different non-variadic prototype than square, from the square call site. It allows all of the other illegal calls.

GCC VTV, and Visual C++ CFG allow all of the illegal calls, even if the non-variadic type signature does not match that of the intended call target.

pi-CFI allows calls to the avg_longs function from the sum_ints indirect call site. avg_longs is address-taken and it has the same static type signature as the intended call target. pi-CFI does not allow illegal calls to non-address-taken functions or functions with different static type signatures. pi-CFI also does not allow calls to variadic functions from non-variadic call sites.

All implementations of CFI allow direct overwrites of variadic arguments, as long as the original control flow of the program is not violated.

7.6.2 Exploit Detection

To evaluate the effectiveness of our tool as a real-world exploit detector, we built a HexVASAN-hardened version of sudo 1.8.3. sudo allows authorized users to execute shell commands as another user, often one with a high privilege level on the system. If compromised, sudo can escalate the privileges of non-authorized users, making it a popular target for attackers. Versions 1.8.0 through 1.8.3p1 of sudo contained a format string vulnerability (CVE-2012-0809) that allowed exactly such a compromise. This vulnerability could be exploited by passing a format string as the first argument (argv[0]) of the sudo program. One such exploit was shown to bypass ASLR, DEP, and glibc's FORTIFY_SOURCE protection [58]. In addition, we were able to verify that GCC 5.4.0 and clang 3.8.0 fail to catch this exploit, even when annotating the vulnerable function with the format function attribute [4] and setting the compiler's format string checking (-Wformat) to the highest level.

Although it is sudo itself that calls the format string function (fprintf), HexVASAN can only detect the violation on the callee side. We therefore had to build hardened versions of not just the sudo binary itself, but also the C library. We chose to do this on the FreeBSD platform, as its standard C library can be easily built using LLVM, and HexVASAN therefore readily fits into the FreeBSD build process. As expected, HexVASAN does detect any exploit that triggers the vulnerability, producing the error message shown in Listing 7.4.

```
$ ln -s /usr/bin/sudo %x%x%x%x
$ ./%x%x%x%x -D9 -A
--------------------------
Error: Index greater than Argument Count
Index is 1
Backtrace:
[0] 0x4053bf <__vasan_backtrace+0x1f> at sudo
[1] 0x405094 <__vasan_check_index+0xf4> at sudo
[2] 0x8015dce24 <__vfprintf+0x2174> at libc.so
[3] 0x8015dac52 <vfprintf_l+0x212> at libc.so
[4] 0x8015daab3 <vfprintf_l+0x73> at libc.so
[5] 0x40bdaf <sudo_debug+0xdf> at sudo
[6] 0x40ada3 <main+0x6c3> at sudo
[7] 0x40494f <_start+0x17f> at sudo
```

Listing 7.4: Exploit detection in sudo.

7.6.3 Variadic Functions Statistics

To collect variadic function usage in real software, we extended our instrumentation mechanism to collect statistics about variadic functions and their calls. As shown in Table 7.2, for each program, we collect:

Call sites. The number of function calls targeting variadic functions. We report the total number and how many of them are indirect, since they are of particular interest for an attack scenario where the adversary can override a function pointer.

Variadic functions. The number of variadic functions. We report their total number and how many of them have their address taken, since CFI mechanism cannot prevent functions with their address taken from being reachable from indirect call sites.

Variadic prototypes. The number of distinct variadic function prototypes in the program.

Functions-per-prototype. The average number of variadic functions sharing the same prototype. This measures how many targets are available, on average, for each indirect call sites targeting a specific prototype. In practice, this the average number of permitted destinations for an indirect call site in the case of a perfect CFI implementation. We report this value both considering all the variadic functions and only those whose address is taken.

Interestingly, each benchmark we analyzed contains calls to variadic functions but few programs (Firefox, OpenSSL, perlbench, gcc, povray, and hmmer) contain indirect calls to variadic functions. In addition to *calling* variadic functions, each benchmark also *defines* numerous variadic functions (421 for Firefox, 794 for Chromium, 1368 for FreeBSD, 469 for Wireshark, and 382 for CPython). Adversaries might be able to modify these calls and to attack the implicit contract

```
static sEnumBuilder _EtherMessageKind("EtherMessageKind",
  JAM_SIGNAL, "JAM_SIGNAL",
  ETH_FRAME, "ETH_FRAME",
  ETH_PAUSE, "ETH_PAUSE",
  ETHCTRL_DATA, "ETHCTRL_DATA",
  ETHCTRL_REGISTER_DSAP, "ETHCTRL_REGISTER_DSAP",
  ETHCTRL_DEREGISTER_DSAP, "ETHCTRL_DEREGISTER_DSAP",
  ETHCTRL_SENDPAUSE, "ETHCTRL_SENDPAUSE",
  0, NULL
);
```

Listing 7.5: Variadic violation in omnetpp.

between caller and callee. However few benchmarks have variadic functions that are called indirectly, often with their address being taken. Moreover, the average number of variadic functions sharing the non-variadic part of the prototype is very little, and is further decreased if we consider only functions that have their address taken. Therefore, in practice, in presence of a strong CFI mechanism, the benefit of HexVASAN is marginal, but necessary to offer a complete protection.

Our sanitizer identified three interesting cases in omnetpp, one of the SPEC CPU2006 benchmarks that implements a discrete event simulator. The benchmark calls a variadic functions with a mismatched type, where it expects a char * but receives a NULL, which has type void *. Listing 7.5 shows the offending code.

We also identified a bug in SPEC CPU2006's perlbench. This benchmark passes the result of a subtraction of two character pointers as an argument to a variadic function. At the call site, this argument is a machine word-sized integer (i.e., 64-bits integer on our test platform). The callee truncates this argument to a 32-bit integer by calling va_arg(list, int). HexVASAN reports this (likely unintended) truncation as a violation, proving its usefulness in finding subtle bugs.

7.6.4 SPEC CPU2006

We measured HexVASAN's run-time overhead by running the SPEC CPU2006 integer (CINT2006) benchmarks on an Ubuntu 14.04.5 LTS machine with an Intel Xeon E5-2660 CPU and 64 GiB of RAM. We ran each benchmark program on its reference inputs and measured the average run-time over three runs. Figure 7.2 shows the results of these tests. We compiled each benchmark with a vanilla clang/LLVM 3.9.1 compiler and optimization level -O3 to establish a baseline. We then compiled the benchmarks with our modified clang/LLVM 3.9.1 compiler to generate the HexVASAN results.

The geometric mean overhead in these benchmarks was just 0.45%, indistinguishable from measurement noise. The only individual benchmark result that stands out is that of libquantum. This benchmark program performed 880M variadic function calls in a run of just 433 seconds.

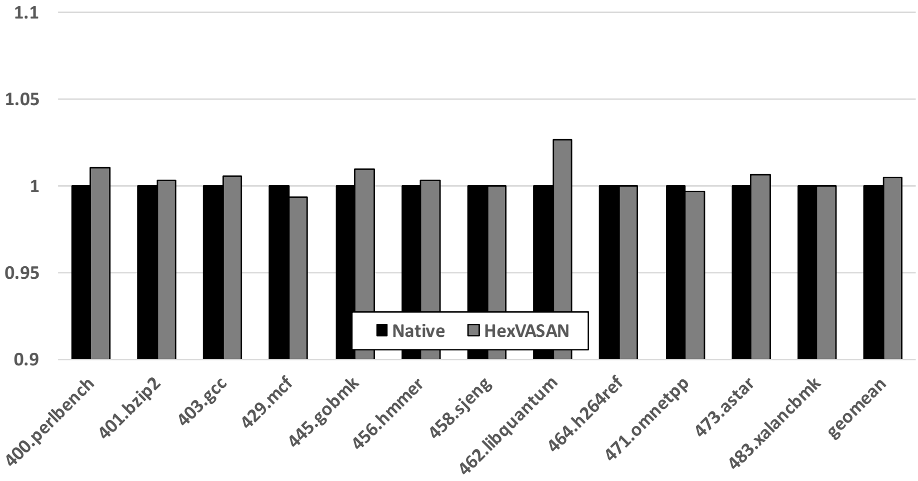


Figure 7.2: Run-time overhead of HexVASAN in the SPECint CPU2006 benchmarks, compared to baseline LLVM 3.9.1 performance.

7.7 Related Works

HexVASAN can either be used as an always-on runtime monitor to mitigate exploits or as a sanitizer to detect bugs, sharing similarities with the sanitizers that exist primarily in the LLVM compiler. Similar to HexVASAN, these sanitizers embed run-time checks into a program by instrumenting potentially dangerous program instructions.

AddressSanitizer [134] (ASan), instruments memory accesses and allocation sites to detect spatial memory errors, such as out-of-bounds accesses, as well as temporal memory errors, such as use-after-free bugs. Undefined Behavior Sanitizer [124] (UBSan) instruments various types of instructions to detect operations whose semantics are not strictly defined by the C and C++ standards, e.g., increments that cause signed integers to overflow, or null-pointer dereferences. Thread Sanitizer [135] (TSAN) instruments memory accesses and atomic operations to detect data races, deadlocks, and various misuses of synchronization primitives. Memory Sanitizer [145] (MSAN) detects uses of uninitialized memory.

CaVer [91] is a sanitizer targeted at verifying correctness of downcasts in C++. Downcasting converts a base class pointer to a derived class pointer. This operation may be unsafe as it cannot be statically determined, in general, if the pointed-to object is of the derived class type. TypeSan [67] is a refinement of CaVer that reduces overhead and improves the sanitizer coverage.

UniSan [93] sanitizes information leaks from the kernel. It ensures that data is initialized before leaving the kernel, preventing reads of uninitialized memory.

All of these sanitizers are highly effective at finding specific types of bugs, but, unlike HexVASAN, they do not address misuses of variadic functions. The aforementioned sanitizers also differ from HexVASAN in that they typically incur significant run-time and memory overhead.

Different control-flow hijacking mitigations offer partial protection against variadic function attacks by preventing adversaries from calling variadic functions through control-flow edges that do not appear in legitimate executions of the program. Among these mitigations, we find Code Pointer Integrity (CPI) [87], a mitigation that prevents attackers from overwriting code pointers in the program, and various implementations of Control-Flow Integrity (CFI), a technique that does not prevent code pointer overwrites, but rather verifies the integrity of control-flow transfers in the program [5, 165, 42, 107, 114, 121, 171, 174, 26, 35, 82, 62, 43, 109, 151, 110, 95, 101, 123, 169, 157, 168, 100, 18, 63, 108, 118].

Control-flow hijacking mitigations *cannot* prevent attackers from overwriting variadic arguments directly. At best, they can prevent variadic functions from being called through control-flow edges that do not appear in legitimate executions of the program. We therefore argue that `HexVASAN` and these mitigations are orthogonal. Moreover, prior research has shown that many of the aforementioned implementations fail to fully prevent control-flow hijacking as they are too imprecise [64, 44, 22, 57], too limited in scope [132, 143], vulnerable to information leakage attacks [56], or vulnerable to spraying attacks [112, 65]. We further showed in Section 7.6.1 that variadic functions exacerbate CFI's imprecision problems, allowing additional leeway for adversaries to attack variadic functions.

Defenses that protect against direct overwrites or misuse of variadic arguments have thus far only focused on format string attacks, which are a subset of the possible attacks on variadic functions. LibSafe detects potentially dangerous calls to known format string functions such as `printf` and `sprintf` [154]. A call is considered dangerous if a `%n` specifier is used to overwrite the frame pointer or return address, or if the argument list for the `printf` function is not contained within a single stack frame. FormatGuard [33] instruments calls to `printf` and checks if the number of arguments passed to `printf` matches the number of format specifiers used in the format string.

Shankar et al. proposed to use static taint analysis to detect calls to format string functions where the format string originates from an untrustworthy source [137]. This approach was later refined by Chen and Wagner [25] and used to analyze thousands of packages in the Debian 3.1 Linux distribution. TaintCheck [104] also detects untrustworthy format strings, but relies on dynamic taint analysis to do so.

`_FORTIFY_SOURCE` of *glibc* provides some lightweight checks to ensure all the arguments are consumed. However, it can be bypassed [3] and does not check for type-mismatch. Hence, none of these aforementioned solutions provide comprehensive protection against variadic argument overwrites or misuse.

7.8 Conclusions

Variadic functions introduce an implicitly defined contract between the caller and callee. When the programmer fails to enforce this contract correctly, the violation leads to runtime crashes or opens up a vulnerability to an attacker. Current tools, including static type checkers and CFI implementations, do not find variadic function type errors or prevent attackers from exploiting calls to variadic functions. Programs such as SPEC CPU2006, Firefox, Apache, CPython, nginx, wireshark and libraries leverage variadic functions to offer flexibility.

We have designed a sanitizer, HexVASAN, that addresses this attack vector. HexVASAN is a light weight runtime monitor that detects bugs in variadic functions and prevents the bugs from being exploited. It imposes negligible overhead (0.45%) on the SPEC CPU2006 benchmarks and is effective at detecting type violations when calling variadic arguments.

	Call sites			Func.			Ratio	
Program	Tot.	Ind.	%	Tot.	A.T.	Proto	Tot.	A.T.
Firefox	30225	1664	5.5	421	18	241	1.75	0.07
Chromium	83792	1728	2.1	794	44	396	2.01	0.11
FreeBSD	189908	7508	3.9	1368	197	367	3.73	0.53
Apache	7121	0	0	94	29	41	2.29	0.71
CPython	4183	0	0	382	0	38	10.05	0.00
nginx	1085	0	0	26	0	14	1.86	0.00
OpenSSL	4072	1	0.02	23	0	15	1.53	0.00
Wireshark	37717	0	0	469	1	110	4.26	0.01
perlbench	1460	1	0.07	60	2	18	3.33	0.11
bzip2	85	0	0	3	0	3	1.00	0.00
gcc	3615	55	1.5	125	0	31	4.03	0.00
mcf	29	0	0	3	0	3	1.00	0.00
milc	424	0	0	21	0	8	2.63	0.00
namd	485	0	0	24	2	8	3.00	0.25
gobmk	2911	0	0	35	0	8	4.38	0.00
soplex	6	0	0	2	1	2	1.00	0.50
povray	1042	40	3.8	45	10	16	2.81	0.63
hmmer	671	7	1	9	1	5	1.80	0.20
sjeng	253	0	0	4	0	3	1.33	0.00
libquantum	74	0	0	91	0	7	13.00	0.00
h264ref	432	0	0	85	5	13	6.54	0.38
lbm	11	0	0	3	0	2	1.50	0.00
omnetpp	340	0	0	48	23	19	2.53	1.21
astar	42	0	0	4	1	4	1.00	0.25
sphinx3	731	0	0	20	0	5	4.00	0.00
xalancbmk	19	0	0	4	2	4	1.00	0.50

Table 7.2: Statistics of Variadic Functions for Different Benchmarks. The second and third columns are variadic call sites broken into "Tot." (total) and "Ind." (indirect); % shows the percentage of variadic call sites. The fifth and sixth columns are for variadic functions. "A.T." stands for *address taken*. "Proto." is the number of distinct variadic function prototypes. "Ratio" indicates the *function-per-prototypes* ratio for variadic functions.

Chapter 8

`leakless`: Bypassing Link-time Hardenings

Throughout the last few decades, computer software has experienced an arms race between exploitation techniques leveraging memory corruption and detection/protection mechanisms. Effective mitigation techniques, such as Address Space Layout Randomization, have significantly increased the difficulty of successfully exploiting a vulnerability. A modern exploit is often two-stage: a first information disclosure step to identify the memory layout, and a second step with the actual exploit. However, because of the wide range of conditions under which memory corruption occurs, retrieving memory layout information from the program is not always possible.

In this paper, we present a technique that uses the dynamic loader's ability to *identify* the locations of critical functions directly and call them, without requiring an information leak. We identified several fundamental weak points in the design of ELF standard and dynamic loader implementations that can be exploited to resolve and execute arbitrary library functions. Through these, we are able to bypass specific security mitigation techniques, including partial and full RELRO, which are specifically designed to protect ELF data-structures from being co-opted by attackers. We implemented a prototype tool, `leakless`, and evaluated it against different dynamic loader implementations, previous attack techniques, and real-life case studies to determine the impact of our findings. Among other implications, `leakless` provides attackers with reliable and non-invasive attacks, less likely to trigger intrusion detection systems.

This chapter is in large parts extracted from [59].

8.1 Introduction

Since the first widely-exploited buffer overflow used by the 1998 Morris worm [113], the prevention, exploitation, and mitigation of memory corruption vulnerabilities have occupied the time of security researchers and cybercriminals alike. Even though the prevalence of memory corruption vulnerabilities has finally begun to decrease in recent years, classic buffer overflows remain the third most common form of software vulnerability, and four other memory corruption vulnerabilities pad out the top 25 [40].

One reason behind the decreased prevalence of memory corruption vulnerabilities is the heavy investment in research on their prevention and mitigation. Specifically, many mitigation techniques have been adopted in two main areas: system-level hardening (such as CGroups [98], AppArmor [13], Capsicum [166], and GRSecurity [61]) and application-level hardening (such as stack canaries [12], Address Space Layout Randomization (ASLR), and the *No-eXecute* (NX) bit [34]).

In particular, *Address Space Layout Randomization* (ASLR), by placing the dynamic libraries in a random location in memory (unknown to the attacker), lead attackers to perform exploits in two stages. In the first stage, the attacker must use an *information disclosure* vulnerability, in which information about the memory layout of the application (and its libraries) is revealed, to identify the address of code that represents security-critical functionality (such as the `system()` library function). In the second stage, the attacker uses a *control flow redirection* vulnerability to redirect the program's control flow to this functionality.

However, because of the wide range of conditions under which memory corruptions occur, retrieving this information from the program is not always possible. For example, memory corruption vulnerabilities in parsing code (e.g., decoding images and video) often take place without a direct line of communication to an attacker, precluding the possibility of an information disclosure. Without this information, performing an exploit against ASLR-protected binaries using current techniques is often infeasible or unreliable.

As noted in [138], despite the race to harden applications and systems, the security of some little-known aspects of application binary formats and the system components using them, have not received much scrutiny. In particular we focus on the *dynamic loader*, a userspace component of the operating system, responsible for loading binaries, and the libraries they depend upon, into memory. Binaries use the dynamic loader to support the *resolution* of imported symbols. Interestingly, this is the exact behavior that an attacker of a hardened application attempts to reinvent by leaking a library's address and contents.

Our insight is that a technique to eliminate the need for an information disclosure vulnerability could be developed by abusing the functionality of the dynamic loader. Our technique leverages weaknesses in the dynamic loader and in the general design of the ELF format to resolve and execute arbitrary library functions, allowing us to successfully exploit hardened applications without the need for an information disclosure vulnerability. Any library function can be executed with this technique, even if it is not otherwise used by the exploited binary, as long as the library that it resides in is loaded. Since almost every binary depends on the C Library, this means our technique allows us to execute security-critical functions such as `system()` and `execve()`, allowing arbitrary command execution. We will also show application-specific library functions can be re-used to perform sophisticated and stealthy attacks. The presented technique is reliable, architecture-agnostic, and does not require the attacker to know the version, layout, content, or any other unavailable information about the library and library function in question.

We implemented our ideas in a prototype tool, called `leakless` [50]. To use `leakless`, the attacker must possess the target application, and have the ability to exploit the vulnerability (i.e., hijack control flow). Given this information, `leakless` can automatically construct an exploit that, without the requirement of an information disclosure, invokes one or more critical library functions of interest.

To evaluate our technique's impact, we performed a survey of several different distributions of Linux (and FreeBSD) and identified that the vast majority of binaries in the default installation of these distributions are susceptible to the attack carried out by `leakless`, if a memory corruption vulnerability is present in the target binary. We also investigated the dynamic loader implementations of various C Libraries, and found that most of them are susceptible to `leakless`' techniques. Additionally, we showed that a popular mitigation technique, RELocation Read-Only (RELRO), which protects library function calls from being redirected by an attacker, is completely bypassable by `leakless`. Finally, we compared the length of `leakless`' ROP chains against ROP compilers implementing similar functionality. `leakless` produces significantly shorter ROP chains than existing techniques, which, as we show, allows it to be used along with a wider variety of exploits than similar attacks created by traditional ROP compilers.

In summary, we make the following contributions:

- We develop a new, architecture- and platform-agnostic attack, using functionality inherent in ELF-based system that supports dynamic loading, to enable an attacker to execute arbitrary library functions without an information disclosure vulnerability.
- We detail, and overcome, the challenges of implementing our system for different dynamic loader implementations and in the presence of multiple mitigation techniques (including RELRO).
- Finally, we perform an in-depth evaluation, including a case study of previously complicated exploits that are made more manageable with our technique, an assessment of the security of several different dynamic loader implementations, a survey of the applicability of our technique to different operating system configurations, and a measurement of the improvement in the length of ROP chains produced by `leakless`.

8.2 Related Works

The memory corruption arms race (i.e., the process of defenders developing countermeasures against known exploit techniques, and attackers coming up with new exploitation techniques to bypass these countermeasures) has been ongoing for several decades. While the history of this race has been documented elsewhere [148], this section focuses on the sequence of events that has required many modern exploits to be *two-stage*, that is, needing an *information disclosure* step before an attacker can achieve arbitrary code execution.

Early buffer overflow exploits relied on the ability to inject binary code (termed *shellcode*) into a buffer, and overwrite a return address on the stack to point into this buffer. Subsequently, when the program would return from its current function, execution would be redirected to the attacker's shellcode, and the attacker would gain control of the program.

As a result, security researchers introduced another mitigation technique: the *NX* bit. The NX bit has the effect of preventing memory areas not supposed to contain code (typically, the stack) from being executed.

The *NX bit* has pushed attackers to adapt the concept of *code reuse*: using functionality already in the program (such as system calls and security-critical library functions) to accomplish their goals. In return-into-libc exploits [122, 153], an attacker redirects the control flow directly to a sensitive libc function (such as `system()`) with the proper arguments to perform malicious behavior, instead of using injected shellcode.

To combat this technique, a system-level hardening technique named *Address Space Layout Randomization* (ASLR) was developed. When ASLR is in place, the attacker does not know the location of libraries, in fact, the program's memory layout (the locations of libraries, the stack, and the heap) is randomized at each execution. Because of this, the attacker does not know *where* in the library to redirect the control flow in order to execute specific functions. Worse, even if the attacker is able to determine this information, he is still unable to identify the location of specific functions inside the library unless he is in possession of a copy of the library. As a result, an attacker usually has to leak the contents of the library itself and parse the code to identify the location of critical functions. To leak these libraries, attackers often reuse small chunks of code (called *gadgets*) in the program's code segment to disclose memory locations. These gadgets are usually combined by writing their addresses onto the stack and consecutively returning to them. Thus, this technique is named *Return Oriented Programming* (ROP) [136].

ROP is a powerful tool for attackers. In fact, it has been shown that a "Turing-complete" set of ROP gadgets can be found in many binaries and can be employed, with the help of a *ROP compiler*, to carry out exploitation tasks [133]. However, because of their generality, ROP compilers tend to produce long ROP chains that, depending on the specific details of a vulnerability, are "too big to be useful" [86]. Later, we will show that `leakless` produces relatively short ROP chains, and, depending on present mitigations, requires very few gadgets. Additionally, `leakless` is able to function without a Turing-complete gadget set.

In real-world exploits, an attacker usually uses an *information disclosure* attack to leak the address or contents of a library, then uses this information to calculate the correct address of a security-critical library function (such as `system()`), and finally sends a second payload to the vulnerable application that redirects the control flow to call the desired function.

In fact, we observed that that the goal of finding the address of a specific library function is actually already implemented by the *dynamic loader*, an OS component that facilitates the resolution of dynamic symbols (i.e., determining the addresses of library functions). Thus, we realized that we could leverage the dynamic loader to remove the information disclosure step, and craft exploits, which would work without the need of an information disclosure attack. Since our attack does not require an information leak step, we call it `leakless`.

The concept of using the dynamic loader as part of the exploitation process was briefly explored in the context of return-into-libc attacks [122, 52, 74]. However, existing techniques are extremely situational [122], platform-dependent, require two stages [74], or are susceptible to current mitigation techniques such as RELRO [122], which we will discuss in future sections. `leakless`, on the other hand, is a single-stage, platform-independent, general technique, and is able to function in the presence of such mitigations.

In the next section, we will describe how the dynamic loader works, and afterwards will show how we abuse this functionality to perform our attack.

8.3 The Dynamic Loader

The dynamic loader is a component of the userspace execution environment that facilitates loading the libraries required by an application at start time and resolving the dynamic symbols (functions or global variables) that are exported by libraries and used by the application. In this section, we will describe how dynamic symbol resolution works on systems based on the ELF binary object specification [131].

ELF is a standard format common to several Unix-like platforms, including GNU/Linux and FreeBSD, and is defined independently from any particular dynamic loader implementation. Since `leakless` mostly relies on standard ELF features, it is easily applicable to a wide range of systems.

8.3.1 The ELF Object

As seen in Section 1.2, an application comprises a main binary ELF file (the executable) and several dynamic libraries, also in ELF format. Each ELF object is composed of *segments*, and each segment holds one or more *sections*.

Each section has a conventional meaning. For instance, the `.text` section contains the code of the program, the `.data` section contains its writeable data (such as global variables), and the `.rodata` section contains the read-only data (such as constants and strings). The list of sections is stored in the ELF file as an array of `Elf_Shdr` structures.

Note that there are two versions of each ELF structure: one version for 32-bit ELF binaries (e.g., `Elf32_Rel`) and one for 64-bit (e.g., `Elf64_Rel`). We ignore this detail for the sake of simplicity, except in specific cases where it is relevant to our discussion.

8.3.2 Dynamic Symbols and Relocations

In this section, we will give a summary of the data structures involved in ELF symbol resolution. Figure 8.1 gives an overview of these data structures and their mutual relationships.

An ELF object can export symbols to and import symbols from other ELF objects. A symbol represents a function or a global variable and is identified by a name. Each symbol is described by a corresponding `Elf_Sym` structure. This struc-

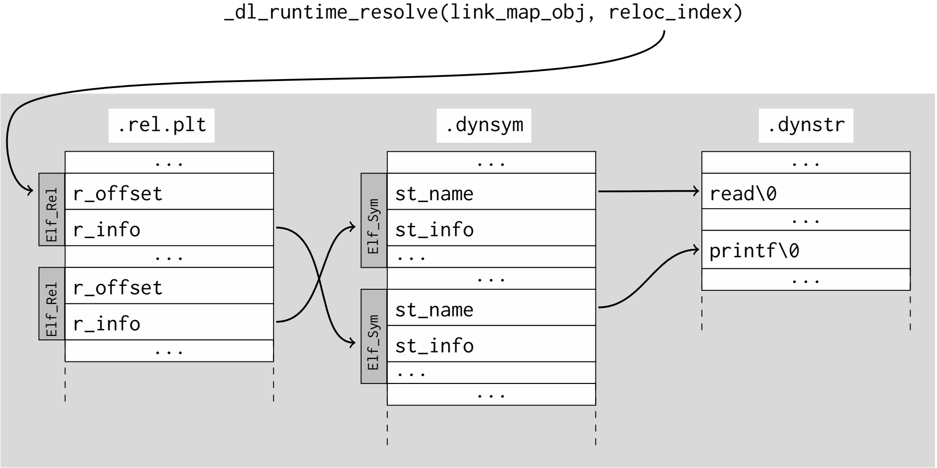


Figure 8.1: The relationship between data structures involved in symbol resolution (without symbol versioning). Shaded background means read only memory.

ture, instances of which comprise the `.dynsym` ELF section, contains the following fields relevant to our work:

`st_name`. An offset, relative to the start of the `.dynstr` section, where the string containing the name of the symbol is located.

`st_value`. If the symbol is exported, the virtual address of the exported function, NULL otherwise.

These structures are referenced to resolve imported symbols. The resolution of imported symbols is supported by relocations, described by the `Elf_Rel` structure. Instances of this structure populate the `.rel.plt` section (for imported functions) and the `.rel.dyn` section (for imported global variables). In our discussion we are only interested to the former section. The `Elf_Rel` structure has the following fields:

`r_info`. The three least significant bytes of this field are used as an unsigned index into the `.dynsym` section to reference a symbol.

`r_offset`. The location (as an absolute address) in memory where the address of the resolved symbol should be written to.

When a program imports a certain function, the linker will include a string with the function's name in the `.dynstr` section, a symbol (`Elf_Sym`) that refers to it in the `.dynsym` section, and a relocation (`Elf_Rel`) pointing to that symbol in the `.rel.plt` section.

The target of the relocation (the `r_offset` field of the `Elf_Rel` struct) will be the address of an entry in a dedicated table: the Global Offset Table (GOT). This table, which is stored in the `.got.plt` section, is populated by the dynamic loader as it resolves the relocations in the `.rel.plt` section.

8.3.3 Lazy Symbol Resolution

Since resolving every imported symbol and applying all relocations at application startup can be a costly operation, symbols are resolved *lazily*. In lazy symbol resolution, the address of a function (which corresponds to an entry in the GOT) is only resolved when necessary (i.e., the first time the imported function is called).

When a program wants to calls an imported function, it instead calls a dedicated stub of code, located in the Procedure Linkage Table (the `.plt` section). As shown in Listing 8.1, each imported function has a stub in the PLT that performs an unconditional indirect jump to the associated entry in the GOT.

After symbol resolution, this GOT entry contains the address of the actual function, in the imported library, and execution continues seamlessly into this function. When the function returns, control flow returns to the *caller* of the PLT stub, and the rest of the PLT stub is not executed. However, at program startup, GOT entries are initialized with an address pointing to the *second* instruction of the associated PLT stub. This part of the stub will push onto the stack an identifier of the imported function (in the form of an offset to an `Elf_Rel` instance in the `.rel.plt` section) and jump to the `PLT0` stub, a piece of code at the beginning of the `.plt` section. In turn, the `PLT0` stub, pushes the value of `GOT[1]` onto the stack and performs an indirect jump to the address of `GOT[2]`. These two entries in the GOT have a special meaning and the dynamic loader populates them at application startup:

`GOT[1]`. A pointer to an internal data structure, of type `link_map`, which is used internally by the dynamic loader and contains information about the current ELF object needed to carry out symbol resolution.

`GOT[2]`. A pointer to a function of the dynamic loader, called `_dl_runtime_resolve`.

In summary, PLT entries basically perform the following function call:

```
_dl_runtime_resolve(link_map_obj, reloc_index)
```

This function uses the `link_map_obj` parameter to access the information it needs to resolve the desired imported function (identified by the `reloc_index` argument) and writes the result into the appropriate GOT entry. After `_dl_runtime_resolve` resolves the imported function, control flow is passed to that function, making the resolution process completely transparent to the caller. The next time the PLT stub for the specified function is invoked execution will be diverted directly to the target function.

The `link_map` structure contains all the information that the dynamic loader needs about a loaded ELF object. Each `link_map` instance is an entry in a doubly-linked list containing the information about all loaded ELF objects.

8.3.4 Symbol Versioning

The ELF standard provides a mechanism to import a symbol with a specific version associated with it. This feature is used to require a function to be imported from

```
100 PLT0:                            196 ; .plt.got start
100    push *0x200                   196 ; Empty entry
106    jmp *0x204                    196 0
110 printf@plt:                      200 ; link_map object
110    jmp *0x208                    200 &link_map_obj
116    push #0                       204 ; Resolver function
11B    jmp PLT0                      204 &_dl_runtime_resolve
120 read@plt:                        208 ; printf entry
120    jmp *0x20C                    208 0x116
126    push #1                       20C ; read entry
12B    jmp PLT0                      20C 0x126
```

Listing 8.1: Example PLT and GOT.

d_tag	d_value
DT_SYMTAB	.dynsym
DT_STRTAB	.dynstr
DT_JMPREL	.rel.plt

d_tag	d_value
DT_PLTGOT	.got.plt
DT_VERNEED	.gnu.version
DT_VERSYM	.gnu.version_r

Table 8.1: Entries of the .dynamic section. d_tag is the key, while d_value is the value.

a specific version of a library. For instance, it is possible to require the fopen C Standard Library function, as implemented in version 2.2.5 of the GNU C Standard Library, using the version identifier GLIBC_2.2.5. The .gnu.version_r section contains version definitions in the form of Elf_Verdef structures.

The association between a dynamic symbol and the Elf_Verdef structure that it refers to is kept in the .gnu.version section, as an array of Elf_Verneed structures, one for each entry in the dynamic symbol table. These structures have a single field: a 16-bit integer that represents an index into the .gnu.version_r section.

Due to this layout, the index in the r_info field of the Elf_Rel structure is used by the dynamic loader as an index into both the .dynsym and .gnu.version sections. This is important to understand, as leakless will later leverage this fact.

8.3.5 The .dynamic Section and RELRO

The dynamic loader collects all the information that it needs about the ELF object from the .dynamic section, which is composed of Elf_Dyn structures. An Elf_Dyn is a key-value pair that stores different types of information. The relevant entries of this section, shown in Table 8.1, hold the absolute addresses of specific sections. One exception is the DT_DEBUG entry, which holds a pointer to an internal data structure of the dynamic loader. This is initialized by the dynamic loader and is used for debugging purposes.

An attacker able to tamper with these values can pose a security risk. For this reason, a protection mechanism known as RELRO (RELocation Read Only) has been introduced in dynamic loaders. RELRO comes in two flavors: partial and full.

Partial RELRO In this mode, some sections, including `.dynamic`, are marked as read-only after they have been initialized by the dynamic loader.

Full RELRO In addition to partial RELRO, lazy resolution is disabled: all import symbols are resolved at startup time, and the `.got.plt` section is completely initialized with the final addresses of the target functions and marked read-only. Moreover, since lazy resolution is not enabled, the `GOT[1]` and `GOT[2]` entries are not initialized with the values we mentioned in Section 8.3.3.

As we will see, RELRO poses significant complications that `leakless` must (and does) address in order to operate in the presence of these countermeasures.

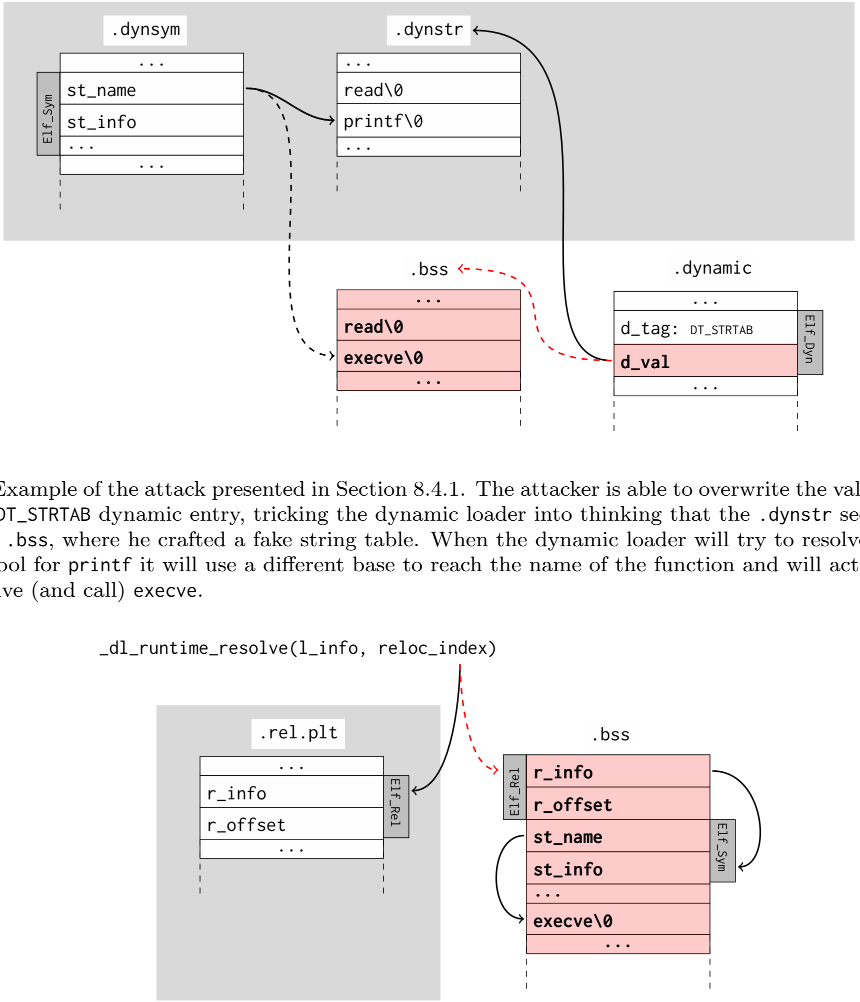
Note that the previously mentioned `link_map` structure stores in the `l_info` field an array of pointers to most of entries in the `.dynamic` section for internal usage. Since the dynamic loader trusts the content of this field implicitly, `leakless` will later be able to misuse this to its own ends.

8.4 The Attack

`leakless` enables an attacker to call arbitrary library functions, using only their name, without any information about the memory layout of the vulnerable program's libraries. To achieve this, `leakless` abuses the dynamic loader, forcing it to resolve and call the requested function. This is possible for the same reason that memory corruption vulnerabilities are so damaging: the mixing of control data and non-control data in memory. In the case of a stack overflow, the control data in question is a stored return address. For the dynamic loader, the control data is comprised of the various data structures that the dynamic loader uses for symbol resolution. Specifically, the *name* of the function, stored in the `.dynstr` section, is analogous to a return address: it specifies a specific target to execute when the function is invoked.

The dynamic loader makes the assumption that the parameters it receives and its internal structures are trustworthy because it assumes that they are provided directly by the ELF file or by itself during initialization. However, when an attacker is able to modify this data, the assumption is broken. Some dynamic loaders (FreeBSD) validate the input they receive. However, they still implicitly trust the control structures, which will be readily corrupted by `leakless`.

`leakless` is designed to be used by an attacker who is attempting to exploit an existing vulnerability. The input to `leakless` is comprised of the executable ELF file, a set of ROP gadgets of the binary (we detail what gadgets an attacker needs in Section 8.5.1), and the name of a library function that the attacker wishes to call (typically, but not necessarily, `execve()`). Given this information, `leakless` outputs a ROP payload that executes the needed library function, bypassing various hardening techniques applied to the binary in question. This ROP chain is generally



(a) Example of the attack presented in Section 8.4.1. The attacker is able to overwrite the value of the DT_STRTAB dynamic entry, tricking the dynamic loader into thinking that the .dynstr section is in .bss, where he crafted a fake string table. When the dynamic loader will try to resolve the symbol for printf it will use a different base to reach the name of the function and will actually resolve (and call) execve.

(b) Example of the attack presented in Section 8.4.2. The reloc_index passed to _dl_runtime_resolve overflows the .rel.plt section and ends up in .bss, where the attacker crafted an Elf_Rel structure. The relocation points to an Elf_Sym located right afterwards overflowing the .dynsym section. In turn the symbol will contain an offset relative to .dynstr large enough to reach the memory area after the symbol, which contains the name of the function to invoke.

Figure 8.2: Illustration of some of the presented attacks. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker.

very short: depending on the mitigations present in the binary, the chain is 3 to 12 write operations. Some examples of the output produced by leakless are available in the documentation of the leakless code repository [50].

leakless does *not* require any information about the addresses or contents of the libraries; we assume that ASLR is enabled for all dynamic libraries and that no knowledge about them is available. However we also assume that the executable is not position-independent, and, thus, is always loaded in a specific location in memory. We discuss this limitation in detail in Section 8.7.2, and show how infrequently Position Independent Executables (PIE) binaries occur in modern OS distributions in Section 8.6.2.

While in most cases, leakless works independently of the dynamic loader implementation and version that the target system is running, some of our attacks require minor modifications to accommodate different dynamic loaders.

Note that leakless's aim, obtaining the address of a library function and call it, is similar to what the dlsym function of libdl does. However, in practice this function is rarely used by applications and, therefore, its address is not generally known to the attacker.

8.4.1 The Base Case

As explained in Section 8.3 and illustrated in Figure 8.1, the dynamic loader starts its work from a Elf_Rel structure in the .rel.plt, then follows the index into the .dynsym section to locate the Elf_Sym structure, and finally uses that to identify the name (a string in the .dynstr section) of the symbol to resolve. The simplest way to call an arbitrary function would be to overwrite the string table entry of an existing symbol with the name of the desired function, and then invoke the dynamic loader, but this is not possible, as the section containing the string table for dynamic symbols, i.e., .dynstr, is not writeable.

However, the dynamic loader obtains the address of the .dynstr section from the DT_STRTAB entry of the .dynamic section, which is at a known location and, by default, writeable. Therefore, as shown in Figure 8.2a, it is possible to overwrite the d_val field of this dynamic entry with a pointer to a memory area under the control of the attacker (typically the .bss or .data section). This memory area would then include a single string, for example execve. At this point, the attacker needs to choose an existing symbol pointing to the correct offset in the fake string table and invoke the resolution of relocation corresponding to that symbol. This can be done by pushing the offset of this relocation on the stack and then jumping to PLT0.

This approach is simple, but it is only effective against binaries in which the .dynamic section is writeable. More sophisticated attacks must be used against binary compiled with partial or full RELRO.

8.4.2 Bypassing Partial RELRO

As we explained in Section 8.3.3, the second parameter of the _dl_runtime_resolve function is the offset of an Elf_Rel entry in the relocation table (.rel.plt section) that corresponds to the requested function. The dynamic loader takes this value and adds it to the base address of the `.rel.plt` to obtain the absolute address of the target `Elf_Rel` structure. However most dynamic loader implementations do not check the boundaries of the relocation table. This means that if a value larger than the size of the `.rel.plt` is passed to `_dl_runtime_resolve`, the loader will use the `Elf_Rel` at the specified location, despite being outside the `.rel.plt` section.

As shown in Figure 8.2b, `leakless` crafts a value for the index that forces the `_dl_runtime_resolve` function to look into a memory area under the control of the attacker. It then crafts an `Elf_Rel` structure that contains, in its `r_offset` field, the address of the writeable memory location where the address of the function will be written. The `r_info` field will, in turn, contain an index that causes the dynamic loader to look into the attacker-controlled memory. `leakless` stores a crafted `Elf_Sym` object at this location, which, likewise, holds a `st_name` field value large enough to point into attacker-controlled memory. Finally, this location is where `leakless` stores the name of the desired function to call.

In sum, `leakless` crafts the full chain of structures involved in symbol resolution, co-opting the process to invoke the function whose name `leakless` has written into attacker-controlled memory. After this, `leakless` pushes the computed offset to the fake `Elf_Rel` structure onto the stack and invokes `PLT0`.

However, this approach is subject to several constraints. First, the symbol index in `Elf_Rel` has to be positive, since the `r_info` field is defined by the ELF standard as an unsigned integer. In practice, this means that the writable memory area (e.g., the `.bss` section) must be located *after* the `.dynsym` section. In our evaluation, this has always been the case.

Another constraint arises when the ELF makes use of the symbol versioning system described in Section 8.3.4. In this case, the `Elf_Rel.r_info` field is not just used as an index into the dynamic symbol table, but also as an index in the symbol version table (the `.gnu.version` section). In general, `leakless` is able to automatically satisfy these constraints, except for x86-64 small binaries using huge pages [127]. We detail the additional constraints introduced by symbol versioning in Appendix A. When the constraints cannot be satisfied, an alternate approach must be adopted. This involves abusing the dynamic loader by corrupting its internal data structures to alter the dynamic resolution process.

8.4.3 Corrupting Dynamic Loader Data

We recall that the first parameter to `_dl_runtime_resolve` is a pointer to a data structure of type `link_map`. This structure contains information about the ELF executable, and the contents of this structure are implicitly trusted by the dynamic loader. Furthermore, `leakless` can obtain the address of this structure from the second entry of the GOT of the vulnerable binary, whose location is deterministically known.

Recall from Section 8.3.5 that the `link_map` structure, in the `l_info` field, contains an array of pointers to the entries of the `.dynamic` section. These are the pointers that the dynamic loader uses to locate the objects that are used during symbol resolution. As shown in Figure 8.3, by overwriting part of this data structure, `leakless` can make the `DT_STRTAB` entry of the `l_info` field point to a specially-crafted dynamic entry which, in turn, points to a fake dynamic string

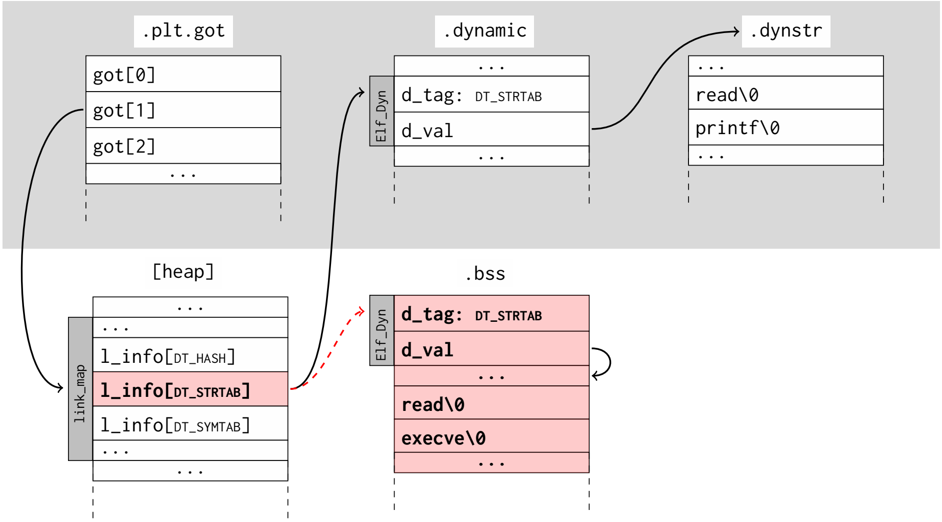


Figure 8.3: Example of the attack presented in Section 8.4.3. The attacker dereferences the second entry of the GOT and reaches the `link_map` structure. In this structure he corrupts the entry of the `l_info` field holding a pointer to the `DT_STRTAB` entry in the dynamic table. Its value is set to the address of a fake dynamic entry which, in turn, points to a fake dynamic string table in the `.bss` section.

table. Hence, the attacker can reduce the situation back to the base case presented in Section 8.4.1.

This technique has wider applicability than the one presented in the previous section, since there are no specific constraints, and, in particular, it is applicable also against small 64 bit ELF binaries using huge pages. However, while in the previous attacks we were relying exclusively on standard ELF features, in this case (and in the one presented in the next section) we assume the layout of a glibc-specific structure (`link_map`) to be known. Each dynamic loader implements this structure in its own way, so minor modifications might be required when targeting a different dynamic loader. Note that `link_map`'s layout might change among versions of the same dynamic loader. However, they tend to be quite stable, and, in particular, in glibc no changes relevant to our attack have taken place since 2004.

8.4.4 The Full RELRO Situation [149]

`leakless` is able to bypass full RELRO protection.

When full RELRO is applied, all the relocations are resolved at load-time, no lazy resolving takes place, and the addresses of the `link_map` structure and of `_dl_runtime_resolve` in the GOT are never initialized. Thus, it is not directly possible to know their addresses, which is what the general technique to bypass

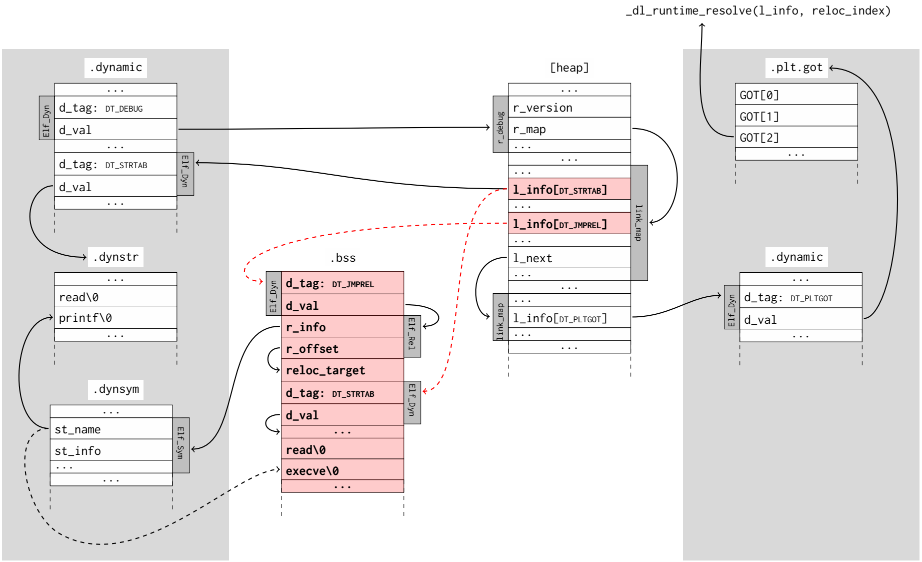


Figure 8.4: Example of the attack presented in Section 8.4.4. Shaded background means read only memory, white background means writeable memory and bold or red means data crafted by the attacker. The attacker goes through the DT_DEBUG dynamic entry to reach the r_debug structure, then, dereferencing the r_map field, he gets to the link_map structure of the main executable, and corrupts l_info[DT_STRTAB] as already seen in Figure 8.3.
Since the .got.plt section is read-only due to full RELRO, the attacker also have to forge a relocation. To do so, he corrupts l_info[DT_JMPREL] making it point to a fake dynamic entry in turn pointing to a relocation. This relocation refers to the existing printf symbol, but has an r_offset pointing to a writeable memory area. Then the attacker also needs to recover the pointer to the _dl_runtime_resolve function, which is not available in the GOT of the main executable due to full RELRO, therefore he dereferences the l_info field of the first link_map structure and gets to the one describing the first shared library, which is not protected by full RELRO. The attacker accesses the l_info[DT_PLTGOT] field and gets to the corresponding dynamic entry (the .dynamic on the right), and then to the .plt.got section (always on the right), at the second entry of which he can find the address of _dl_runtime_resolve.

partial RELRO relies upon.

However, it is possible to indirectly recover these two values through the DT_DEBUG entry in the dynamic table. The value of the DT_DEBUG entry is set by the dynamic loader at load-time to point to a data structure of type r_debug. This data structure contains information used by debuggers to identify the base address of the dynamic loader and to intercept certain events related to dynamic loading. In addition, the r_map field of this structure holds a pointer to the head of the linked list of link_map structures.

leakless corrupts the first entry of the list describing the ELF executable so that the l_info entry for DT_STRTAB points to a fake dynamic string table. This is presented in Figure 8.4.

After this, leakless must invoke _dl_runtime_resolve, passing the link_map structure that it just corrupted as the first argument and an offset into the new .dynsym as the second parameter. However _dl_runtime_resolve is not available in the GOT due to full RELRO. Therefore, leakless must look for its address in the GOT of *another* ELF object, namely, a library loaded by the application that is not protected by full RELRO. In most cases, only ELF executables are compiled with full RELRO, and libraries are not. This is due to the fact that RELRO is designed to harden, at the cost of performance, specific applications that are deemed "risky". Applying full RELRO to a shared library would impact the performance of all applications making use of this library, and thus, libraries are generally left unprotected. Since the order of libraries in the linked list is deterministic, leakless can dereference the l_next entry in link_map to reach the entry describing a library that is not protected by full RELRO, dereference the entry in l_info corresponding to the DT_PLTGOT dynamic entry, dereference its value (i.e., the base address of that library's GOT), and read the address of _dl_runtime_resolve from this GOT.

leakless must then overcome a final issue: _dl_runtime_resolve will not only call the target function, but will also try to write its address to the appropriate GOT entry. If this happens, the program will crash, as the GOT is read-only when full RELRO is applied. We can circumvent this issue by faking the DT_JMPREL dynamic entry in the link_map structure that points to the .rel.dyn section. leakless points it to an attacker-controlled memory area and writes an Elf_Rel structure, with a target (r_offset field) pointing to a writeable memory area, referring to the symbol we are targeting. Therefore, when the library is resolved, the address will be written to a writeable location, the program will not crash, and the requested function will be executed.

8.5 Implementation

leakless analyzes a provided binary to identify which of its techniques is applicable, crafts the necessary data structures, and generates a ROP chain that implements the chosen technique. The discovery of the initial vulnerability itself, and the automatic extraction of usable gadgets from a binary are orthogonal to the scope of our work, and have been well-studied in the literature and implemented in the real world [68, 24, 133, 150, 70, 53]. We designed leakless to be compatible with a number of gadget finding techniques, and have implemented a manual back-end (where gadgets are provided by the user) and a back-end that utilizes ROPC [86], an automated ROP compiler prototype built on the approach proposed by Q [133].

We also developed a small test suite, composed of a small C program with a stack-based buffer overflow compiled, alternatively, with no protections, partial RELRO, and full RELRO. The test suite runs on GNU/Linux with the x86, x86-64 and ARM architectures and with FreeBSD x86-64.

8.5.1 Required Gadgets

`leakless` comprises four different techniques that are used depending on the hardening techniques applied to the binary. These different techniques require different gadgets to be provided to `leakless`. A summary of the types of gadgets is presented in Table 8.2. The `write_memory` gadget is mainly used to craft data structures at known memory locations, while the `deref_write` gadget to traverse and corrupt data structures (in particular `link_map`). The `deref_save` and `copy_to_stack` gadgets are used only in the full RELRO case. The aim of the former is to save at a known location the address of `link_map` and `_dl_runtime_resolve`, while the latter is used to copy `link_map` and the relocation index on the stack before calling `_dl_runtime_resolve`, since using `PLT0` is not a viable solution.

For the interested reader, we provide in-depth examples of executions of `leakless` in the presence of two different sets of mitigation techniques in the documentation of the `leakless` code repository [50].

8.6 Evaluation

We evaluated `leakless` in four ways. First, we determined the applicability of our technique against different dynamic loader implementations. We then analyzed the binaries distributed by several popular GNU/Linux and BSD distributions (specifically, Ubuntu, Debian, Fedora, and FreeBSD) to determine the percentage of binaries that would be susceptible to our attack. Then we applied `leakless` in two real-world exploits against a vulnerable version of Wireshark and in a more sophisticated attack against Pidgin. Finally we used a Turing-complete ROP compiler to implement the approach used in `leakless` and two other previously used techniques, and compared the size of the resulting chains.

8.6.1 Dynamic Loaders

To show `leakless`' generality, especially across different ELF-based platforms, we surveyed several implementations of dynamic loaders. In particular, we found that the dynamic loader part of the *GNU C Standard Library* (also known as glibc and widely used in GNU/Linux distributions), several other Linux implementations such as *dietlibc*, *uClibc* and *newlib* (widespread in embedded systems) and the *OpenBSD* and *NetBSD* implementations are vulnerable to `leakless`. Another embedded library, *musl*, instead, is not susceptible to our approach since it does not support lazy loading. *Bionic*, the C Standard Library used in Android, is also not

Signature	Implementation	RELRO			
		N	P	H	F
write_memory $(destination, value)$	$*(destination) = value$	✓	✓	✓	✓
deref_write$(pointer, offset, value)$	$*(*(pointer) + offset) = value$			✓	✓
deref_save $(destination, pointer, offset)$	$*(destination) = *(*(pointer) + offset)$				✓
copy_to_stack $(offset, source)$	$*(stack\_pointer + offset) = *(source)$				✓

Table 8.2: Gadgets required for the various approaches. The “Signature” column represents the name of the gadget and the parameters it accepts, while “Implementation” presents the behavior of the gadget in C-like pseudo code. The last four columns indicate whether a certain gadget is required for the corresponding approach presented in Section 8.4. Under RELRO, “N” indicates RELRO is disabled, “P” means partial RELRO is used, “H” stands for the partial RELRO and small 64 bit binaries using huge pages, and “F” denotes that full RELRO is enabled.

vulnerable since it only supports PIE binaries. The most interesting case, out of all the loaders we analyzed, is *FreeBSD*'s implementation. In fact, it is the only one which performs boundary checks on arguments passed to `_dl_runtime_resolve`. All other loaders implicitly trust input arguments argument. Furthermore, *all* analyzed loaders implicitly trust the control structures that `leakless` corrupts in the course of most of its attacks.

In summary, out of all of the loaders we analyzed, only two are immune to `leakless` by design: musl, which does not support lazy symbol resolution, and bionic, which only supports PIE executables. Additionally, because the FreeBSD dynamic loader performs bounds checking, the technique explained in Section 8.4.2 is not applicable. However, the other techniques still work.

8.6.2 Operating System Survey

To understand `leakless`' impact on real-world systems, we performed a survey of all binaries installed in default installations of several different Linux and BSD distributions. Specifically, we checked all binaries in `/sbin`, `/bin`, `/usr/sbin`, and `/usr/bin` on these systems and classified the binaries by the applicability of the techniques used by `leakless`. The distributions that we considered were Ubuntu 14.10, Debian Wheezy, Fedora 20, and FreeBSD 10. We used both x86 and x86-64 versions of these systems. On Ubuntu and Debian, we additionally installed the LAMP (Linux, Apache, MySQL, PHP) stack as an attempt to simulate a typical server deployment configuration.

The five categories that we based our ratings on are as follows:

Unprotected. This category includes binaries that have no RELRO or PIE. For these binaries, `leakless` can apply its base case technique, explained in Section 8.4.1.

Partial RELRO. Binaries that have partial RELRO, but lack PIE, fall into this category. In this case, `leakless` would apply the technique described in Section 8.4.2.

Partial RELRO (huge pages). Binaries in this category have partial RELRO, use huge pages, and are very small, therefore, they require `leakless` to use the technique described in Section 8.4.3. They are included in this category.

Full RELRO. To attack binaries that use full RELRO, which comprise this category, `leakless` must apply the technique presented in Section 8.4.4.

Not susceptible. Finally, we consider binaries that use PIE to be insusceptible to `leakless` (further discussion on this in Section 8.7.2).

The results of the survey, normalized to the total number of binaries in an installation, are presented in Figure 8.5. We determined that, on Ubuntu, 84% of the binaries were susceptible to at least one of our techniques and 16% were protected with PIE. On Debian, `leakless` can be used on 86% of the binaries. Fedora has 76% of susceptible binaries. Interestingly, FreeBSD ships no binaries with RELRO or PIE, and is thus 100% susceptible to `leakless`.

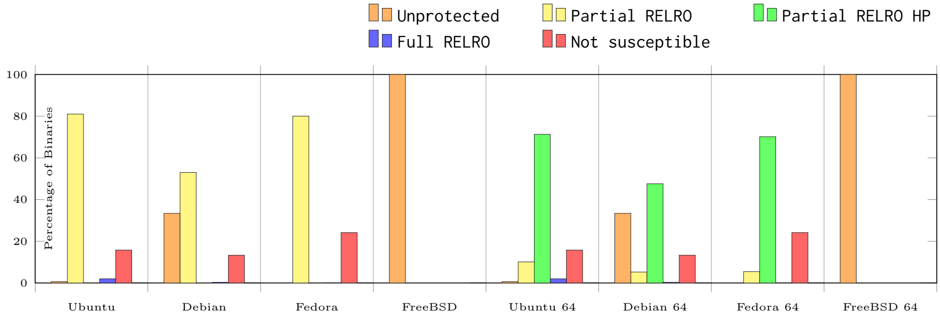


Figure 8.5: Classification of the binaries in default installations of target distributions. Binaries marked as `Unprotected`, `Partial RELRO`, `Partial RELRO HP` and `Full RELRO` require, respectively, to the attacks detailed in Section 8.4.1, Section 8.4.2, Section 8.4.3 and Section 8.4.4, while for `Not susceptible` binaries, the `leakless` approach is not applicable

Additionally, we performed a survey on the shared libraries of the systems we considered. We found that, on average, only 11% of the libraries had full RELRO protection. This has some interesting implications for `leakless`: for a given binary, the likelihood of finding a loaded library without full RELRO is extremely high and, even if a vulnerable binary employs RELRO, `leakless` can still apply its full RELRO attack to bypass this. This has the effect of making RELRO basically useless as a mitigation technique, unless it is applied system-wide.

8.6.3 Case Study: Wireshark

We carried out a case study in applying `leakless` to a vulnerability in a program that does not present a direct line of communication to the attacker. In other words, the exploit must be done in one-shot, with no knowledge of the layout of the address space or the contents of libraries.

We picked a recent (April 2014) vulnerability [31], which is a stack-based buffer overflow in *Wireshark*'s MPEG protocol parser in versions 1.8.0 through 1.8.13 and 1.10.0 through 1.10.6. We carried out our experiments against a Wireshark 1.8.2 binary compiled with partial RELRO and one compiled with full RELRO. Both were compiled for x86-64 on Debian Wheezy and used the GNU C Library, without other protections such as PIE and stack canaries.

We used the manual `leakless` back-end to identify the required gadgets to construct the four necessary primitives (described in Section 8.5.1): *write_memory*, *deref_write*, *deref_save* and *copy_to_stack*. In the case of Wireshark, it was trivial to find gadgets to satisfy all of these primitives.

`leakless` was able to construct a one-shot exploit using the attacks presented in Section 8.4.2 and Section 8.4.4. In both cases, the exploit leverages the dynamic loader in order to call the `execve` function from glibc to launch an executable of our choice.

```c
void *p, *a;
p = purple_proxy_get_setup(0);
purple_proxy_info_set_host(p, "legit.com");
purple_proxy_info_set_port(p, 8080);
purple_proxy_info_set_type(p, PURPLE_PROXY_HTTP);

a = purple_accounts_find("usr@xmpp", "prpl-xmpp");
purple_account_disconnect(a);
purple_account_connect(a);
```

Listing 8.2: The Pidgin attack.

8.6.4 Case Study: Pidgin

We also applied `leakless` to Pidgin, a popular multi-protocol instant-messaging client, to build a more sophisticated exploit. Specifically, we wanted to perform a malicious operation without calling any anomalous system call which could trigger intrusion detection systems (e.g., `execve("/bin/sh")`). We used Pidgin 2.10.7, building it from the official sources with RELRO enabled and targeting the x86 architecture.

To this end, we crafted an exploit designed to masquerade itself in legitimate functionality present in the application logic: tunneling connections through a proxy. The idea of the attack is that an IM service provider exploits a vulnerability such as CVE-2013-6487 [41] to gain code execution, and, using Pidgin's global proxy settings, redirects all IM traffic through a third-party server to enable chat interception.

Once we identified the necessary gadgets to use `leakless` with full RELRO protection, it was easy to invoke functions contained in `libpurple.so` (where the core of the application logic resides) to perform the equivalent of the C code shown in Listing 8.2.

Interestingly, some of this library-provided functionality is not imported into the Pidgin executable itself, and would be very challenging to accomplish in a single-stage payload, without `leakless`.

8.6.5 ROP Chain Size Comparison

To prove the effectiveness of the `leakless` approach, we compared it with two existing techniques that allow an attacker to call arbitrary library functions. The first consists in scanning a library backwards, starting from an address in the `.plt.got` section, until the ELF header is found, and then scan forward to find a fingerprint of the function the attacker wants to invoke. This approach is feasible, but not very reliable, since different versions (or implementations) of a library might not be uniquely identified with a single fingerprint. The second technique is more reliable, since it implements the full symbol resolution process, as it is carried out by the dynamic loader.

Technique	First call	Subsequent	Feasibility
ROPC - scan library	3468 bytes	+340 bytes	16.38%
ROPC - symbol resolution	7964 bytes	+580 bytes	8.67%
leakless partial RELRO	648 bytes	+84 bytes	73.78%
leakless full RELRO	2876 bytes	+84 bytes	17.44%
leakless* partial RELRO	292 bytes	+48 bytes	95.24%
leakless* full RELRO	448 bytes	+48 bytes	88.9%

Table 8.3: Size of the ROP chains generated by ROPC for each technique presented in Section 8.6.5, and by leakless' manual back-end (*). The second column represents the size in bytes for the setup and the first call, while the third column shows the additional cost (in bytes) for each subsequent call. Finally, the fourth column indicates the percentage of vulnerabilities used in Metasploit that would be feasible to exploit with a ROP chain of the *First call* size.

We implemented these two approaches using a Turing-complete ROP compiler for x86, based on Q [133], called ROPC [86]. We compare these approaches against that of leakless' ROPC back-end, in partial RELRO and full RELRO modes. For completeness, we also include the leakless' manual back-end, with gadgets specified by the user.

In fact, the size of a ROP payload is critical, vulnerabilities often involve an implicit limit on the size of the payload that can be injected into a program. To measure the impact of leakless' ROP chain size, we collected the size limits imposed on payloads of all the vulnerability descriptions included in the *Metasploit Framework* [126], a turn-key solution for launching exploits against known vulnerabilities in various software. We found that 946 of the 1,303 vulnerability specifications included a maximum payload size, with an average specified maximum payload size of 1332 bytes. To demonstrate the increase in the feasibility of automatically generating complex exploits, we include, for each evaluated technique, the percentage of Metasploit vulnerabilities for which the technique can automatically produce short enough ROP chains.

The results, in terms of length of the ROP chain generated for ROPC's test binaries and feasibility against the vulnerabilities used in Metasploit, are shown in Table 8.3. leakless outperforms existing techniques, not only in the absolute size of the ROP chain to perform the initial call, but also in the cost of performing each additional call, which is useful in a sophisticated attack such as the one demonstrated in Section 8.6.4.

8.7 Discussion

In this section, we will discuss several aspects relating to leakless: why the capabilities that it provides to attackers are valuable, when it is most applicable, what its limitations are, and what can be done to mitigate against them.

8.7.1 leakless Applications

leakless represents a powerful tool in the arsenal of exploit developers, aiding them in three main areas: functionality reuse, one-shot exploitation, and ROP chains shortening.

One-shot exploitation. While almost any exploit can be simplified by leakless, we have designed it with the goal of enabling exploits that, without it, require an information disclosure vulnerability, but for which an information disclosure is not feasible or desirable. A large class of programs that fall under this category are file format parsers.

Code that parses file formats is extremely complex and, due to the complex, untrusted input that is involved, this code is prone to memory corruption vulnerabilities. There are many examples of this: the image parsing library libpng had 27 CVE entries over the last decade [37], and libtiff had 53 [38]. Parsers of complex formats suffer even more: the multimedia library ffmpeg has accumulated 170 CVE entries over the last five years alone [36]. This class of libraries is not limited to multimedia. Wireshark, a network packet analyzer, has 285 CVE entries, most of which are vulnerabilities in network protocol analysis plugins [39].

These libraries, and others like them, are often used *offline*. The user might first download a media or PCAP file, and *then* parse it with the library. At the point where the vulnerability triggers, an attacker cannot count on having a direct connection to the victim to receive an information disclosure and send additional payloads. Furthermore, most of these formats are *passive*, meaning that (unlike, say, PDF), they cannot include scripts that the attacker can use to simulate a two-step exploitation. As a result, even though these libraries might be vulnerable, exploits for them are either extremely complex, unreliable, or completely infeasible. By avoiding the information disclosure step, leakless makes these exploits simpler, reliable, and feasible.

Functionality reuse. leakless empowers attackers to call arbitrary functions from libraries loaded by the vulnerable application. In fact, the vulnerable application does not have to actually *import* this function; it just needs to link against the library (i.e., call any other function in the library). This is brings several benefits.

To begin with, the C Standard Library, which is linked against by most applications, includes functions that wrap almost every system call (e.g., read(), execve(), and so on). This means that leakless can be used to perform any system call, in a short ROP chain, even without a system call gadget.

Moreover, as demonstrated in Section 8.6.4, leakless enables easy reuse of existing functionality present in the application logic. This is important for two reasons.

First, this helps an attacker perform stealthy attacks by making it easier to masquerade an exploit as something the application might normally do. This can be crucial when a standard exploitation path is made infeasible by the presence of protection mechanisms such as seccomp [9], AppArmor [2], or SELinux [103].

Second, depending on the goals of the attacker, reusing program functionality may be better than simply executing arbitrary commands. Aside from the attack discussed in our Pidgin case study, an attacker can, for example, silently enable insecure cipher-suites, or versions of SSL, in the Firefox web browser with a single function call to `SSL_CipherPrefSetDefault` [102].

Shorter ROP chains. As demonstrated in Section 8.6.5, `leakless` produces shorter ROP chains than existing techniques. In fact, in many cases, `leakless` is able to produce ROP chains less than one kilobyte that lead to the execution of arbitrary functions. As many vulnerabilities have a limit as to the maximum size of the input that they will accept, this is an important result. For example, the vulnerability that we exploited in our Pidgin case study allowed a maximum ROP chain of one kilobyte. Whereas normal ROP compilation techniques would be unable to create automatic payloads for this vulnerability, `leakless` was able to call arbitrary functions via an automatically-produced ROP chain that remained within the length limit.

8.7.2 Limitations

`leakless`' biggest limitation is the inability to handle Position Independent Executables (PIEs) without a prior information disclosure. This is a general problem to any technique that uses ROP, as the absolute addresses of gadgets must be provided in the ROP chain. Additionally, without the base address of the binary, `leakless` would be unable to locate the dynamic loader structures that it needs to corrupt.

When presented with a PIE executable, `leakless` requires the attacker to provide the application's base address, which is presumably acquired via an information disclosure vulnerability (or, for example, by applying the technique presented in BROP [17]). While this breaks `leakless`' ability to operate without an information disclosure, `leakless` is likely still the most convenient way to achieve exploitation, as no library locations or library contents have to be leaked. Additionally, depending on the situation, the disclosure of just the address of the binary might be more feasible than the disclosure of the *contents* of an entire library. Unlike other techniques, which may need the latter, `leakless` only requires the former.

In practice, PIEs are uncommon due to the associated cost in terms of performance. Specifically, measurements have shown that PIE overhead on x86 processors averages at 10%, while the overhead on x86-64 processors, thanks to instruction-pointer-relative addressing, averages at 3.6% [117].

Because of the overhead associated with PIE, most distributions ship with PIE enabled only for those applications deemed "risky". For example, according to their documentation, Ubuntu ships only 27 of their officially supported packages (i.e., packages in the "main" repository) with PIE enabled, out of over 27,000 packages [155]. As shown in Section 8.6.1, PIE executables comprise a minority of the executables on all of the systems that we surveyed.

8.7.3 Countermeasures

There are several measures that can be taken against `leakless`, but they all have drawbacks. In this sections we analyze the most relevant ones.

Position Independent Executables. A quick countermeasure is to make every executable on the system position independent. While this would block leakless's automatic operation (as discussed in Section 8.7.2), it would still allow the application of the leakless technique when any information disclosure does occur. For that reason, and the performance overhead associated with PIE, we consider the other countermeasures described in this section to be better solutions to the problem.

Disabling lazy loading. When the LD_BIND_NOW environment variable is set, the dynamic loader will completely disable *lazy loading*. That is, all imports, for the program binary and any library it depends on, are resolved upon program startup. As a side-effect of this, the address of _dl_runtime_resolve does not get loaded into the GOT of any library, and leakless cannot function. This is equivalent to enable full RELRO on the whole system, and consequently, it incurs in the same, non-negligible, performance overhead.

Disabling DT_DEBUG. Finally, leakless also uses the DT_DEBUG dynamic entry, used by debuggers for intercepting loading-related events, to bypass full RELRO. Currently, this field is always initialized, opening the doors for leakless' full RELRO bypass. To close this hole, the dynamic loader could be modified to only initialize this value when a debugger is present or in the presence of an explicitly-set environment variable.

Better protection of loader control structures. leakless heavily relies on the fact that dynamic loader control structures are easily accessible in memory, and their locations are well-known. It would be beneficial for these structures to be better protected, or hidden in memory, instead of being loaded at a known location. For example, as shown in [120], these structures, along with any sections that provide control data for symbol resolution, could be marked as read-only after initialization. Such a development would eliminate leakless' ability to corrupt these structures and would prevent the attack from redirecting the control flow to sensitive functions.

Additionally, modifying the loading procedure to use a table of link_map structure, and letting _dl_runtime_resolve take an index in this table, instead of a direct pointer, will break leakless' bypass of full RELRO. However, this change would also break compatibility with any binaries compiled before the change is implemented.

Isolation of the dynamic loader. Isolating the dynamic loader from the address space of the target program could be an effective countermeasure. For instance, on Nokia's *Symbian OS*, which has a micro-kernel, the dynamic loader is implemented in a separate process as a *system server* which interfaces with the kernel[111]. This guarantees that the control structures of the dynamic loader cannot be corrupted by the program, and, therefore, this makes leakless virtually ineffective. However, such a countermeasure would have a considerable impact on the overall performance of applications due to the overhead of IPC (Inter-Process Communication).

In general, the mitigations either represent a runtime performance overhead (PIE or loader isolation), a load-time performance overhead (non-lazy loading and system-wide RELRO), or a modification of the loading process (DT_DEBUG disabling or loader control structure hiding). In the long run, we believe that a redesign of the dynamic loader, with security in mind, would be extremely beneficial to the community. In the short term, there are options available to protect against `leakless`, but they all come with a performance cost.

8.8 Conclusion

In this paper, we presented `leakless`, a new technique that leverages functionality provided by the dynamic loader to enable attackers to use arbitrary, security-critical library functions in their exploits, without having to know where in the application's memory these functions are located. This capability allows exploits that, previously, required an information disclosure step to function.

Since `leakless` leverages features mandated in the ELF binary format specification, the attacks it implements are applicable across architectures, operating systems, and dynamic loader implementations. Additionally, we showed how our technique can be used to bypass hardening schemes such as RELRO, which are designed to protect important control structures used in the dynamic resolution process. Finally, we proposed several countermeasures against `leakless`, discussing the advantages and disadvantages of each one.

Conclusions

In this work, we focused on the design and implementation of binary analysis techniques that are general enough to be applicable to a large set of different instruction sets and ABIs. There's virtually no limit on the tools that can be built on top of this platform.

The next step is to improve the performance of the generated code (Section 2.4) and assess the quality of the argument and return value detection algorithm proposed in Chapter 5.

In the future, our plan is to build an UI for `rev.ng` providing fine grained and accurate information about the code, featuring symbolic execution (thanks to KLEE [21]), a timeless debugger similar to QIRA [73], a fuzzer for binary programs integrated with AFL [90], and most importantly a decompiler emitting C code that can be modified and able to apply the changes in the original binary.

Before getting there it's vital to develop an accurate algorithm to recover high-level control constructs (such as `if` statements and `while` loops), an undergoing effort based on existing work [167]. This, along with detection of local variables, which can be obtained as a side effect of the *stack analysis* (Section 5.2.2), will allows us to build a full-fledged interactive decompiler able to handle many architectures with ease.

Bibliography

[1] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8617.

[2] AppArmor. http://wiki.apparmor.net/.

[3] A Eulogy for Format Strings. http://phrack.org/issues/67/9.html.

[4] Using the GNU Compiler Collection (GCC) - Function Attributes. https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Function-Attributes.html.

[5] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[6] Continuum Analytics. llvmpy. http://www.llvmpy.org/.

[7] D.A. Andriesse, J.M. Slowinska, and H.J. Bos. *Compiler-Agnostic Function Detection in Binaries*, volume EuroS&P. IEEE Computer Society, 2017 edition, 4 2017.

[8] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, Austin, TX, 2016. USENIX Association.

[9] Andrea Arcangeli. seccomp. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[10] Gogul Balakrishnan and Thomas Reps. *Compiler Construction: 13th Int. Conf., CC 2004*, chapter Analyzing Memory Accesses in x86 Executables, pages 5–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[11] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 845–860, Berkeley, CA, USA, 2014. USENIX Association.

[12] Arash Baratloo, Navjot Singh, and Timothy K Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *USENIX Annual Technical Conference, General Track*, pages 251–262, 2000.

[13] Mick Bauer. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 2006(148):13, 2006.

[14] 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.

[15] Priyam Biswas. HexVASAN. `https://github.com/HexHive/HexVASAN`.

[16] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. Venerable Variadic Vulnerabilities Vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 186–198, Vancouver, BC, 2017. USENIX Association.

[17] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[18] Dimitar Bounov, Rami Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[19] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[20] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] 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.

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[23] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-granularity Software Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[24] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[25] Karl Chen and David Wagner. Large-scale analysis of format string vulnerabilities in debian linux. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007.

[26] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert Huijie Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[27] Moo-Kyoung Chung and Chong-Min Kyung. Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time. In *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, pages 38–44, June 2004.

[28] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *xProc. of the 7th Int. Workshop on Program Comprehension*, IWPC '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society.

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

[30] Lucian Cojocar. Commit fixing the memset bug in uClibc-ng, 2016. `http://bit.ly/2cx2Lp2`.

[31] Common Vulnerabilities and Exposures. CVE-2014-2299. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299`.

[32] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[33] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *USENIX Security Symposium*, 2001.

[34] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[35] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[36] CVEDetails.com. ffmpeg: CVE security vulnerabilities. `http://www.cvedetails.com/product/6315/Ffmpeg-Ffmpeg.html`.

[37] CVEDetails.com. Libpng: Security Vulnerabilities. http://www.cvedetails.com/vendor/7294/Libpng.html.

[38] CVEDetails.com. Libtiff: CVE security vulnerabilities. http://www.cvedetails.com/product/3881/Libtiff-Libtiff.html.

[39] CVEDetails.com. Wireshark: CVE security vulnerabilities. http://www.cvedetails.com/product/8292/Wireshark-Wireshark.html.

[40] CWE. CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[41] National Vulnerability Database. NVD - Detail - CVE-2013-6487. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6487.

[42] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[43] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Annual Design Automation Conference (DAC)*, 2014.

[44] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[45] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing&Trade; Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proc. of the Int. Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[46] The Valgrind Developers. Valgrind. http://valgrind.org/.

[47] Alessandro Di Federico. llvmcpy. https://github.com/revng/llvmcpy.

[48] Alessandro Di Federico. rev.ng. https://rev.ng/.

[49] Alessandro Di Federico and Giovanni Agosta. A Jump-target Identification Method for Multi-architecture Static Binary Translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 17:1–17:10, New York, NY, USA, 2016. ACM.

[50] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Leakless source code repository. https://github.com/ucsb-seclab/leakless.

[51] 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.

[52] Sébastien Dudek. The Art Of ELF: Analysis and Exploitations. `http://bit.ly/1a8MeEw`.

[53] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In *WOOT*, 2010.

[54] Erik Andersen. uClibc, 2012. `https://www.uclibc.org/`.

[55] Doug Evans, Frank Ch. Eigler, Ben Elliston, and Dave Brolley. CGEN. `http://www.sourceware.org/cgen/`.

[56] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[57] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[58] Exploit Database. sudo_debug privilege escalation. `https://www.exploit-db.com/exploits/25134/`, 2013.

[59] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF Ruined Christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, Washington, D.C., 2015. USENIX Association.

[60] Philip J. Fleming and John J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM*, 29(3):218–221, March 1986.

[61] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements. 2009.

[62] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[63] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symp. on Security and Privacy*, 2016.

[64] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[65] Enes Göktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (And What to do About it). In *USENIX Security Symposium*, 2016.

[66] GrammaTech, Inc. CodeSurfer. `http://bit.ly/1TGy7u2`.

[67] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. TypeSan: Practical Type Confusion Detection. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[68] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security*, pages 49–64, 2013.

[69] Laune C. Harris and Barton P. Miller. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News*, 33(5):63–68, December 2005.

[70] Christian Heitman and Ivan Arce. BARFgadgets. `https://github.com/programa-stic/barf-project/tree/master/barf/tools/gadgets`.

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

[72] Hex-Rays. IDA Pro. `http://bit.ly/1gybdzm`, retrieved Feb. 2016.

[73] George Hotz. QIRA. `http://qira.me/`.

[74] inaz2. ROP Illmatic: Exploring Universal ROP on glibc x86-64. `http://ja.avtokyo.org/avtokyo2014/speakers#inaz2`.

[75] Anaconda Inc. llvmlite. `http://llvmlite.pydata.org/en/latest/`.

[76] Qualcomm Technologies Inc. Hexagon SDK - DSP Processor. `https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor`.

[77] Synopsys Inc. Synopsys DesignWare ARC Configurable Processor Cores. `https://www.synopsys.com/designware-ip/processor-solutions/arc-processors.html`.

[78] Intel. Intel Active Management Technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-active-management-technology.html`.

[79] Intel Corp. Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[80] Information technology – Programming languages – C++. Standard, International Organization for Standardization, Geneva, CH, December 2014.

[81] Information technology – Programming languages – C. Standard, International Organization for Standardization, Geneva, CH, December 2011.

[82] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[83] Jakub Jelinek. FORTIFY_SOURCE. `https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html`, 2004.

[84] Johannes Kinder. Jakstab. `http://www.jakstab.org/`.

[85] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] Paul Kot. A Turing complete ROP compiler. `https://github.com/pakt/ropc`.

[87] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[88] J. Křoustek. Retargetable Analysis of Machine Code. page 190, 2015.

[89] 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.

[90] lcamtuf. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

[91] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *USENIX Security Symposium*, 2015.

[92] Linux Programmer's Manual. va_start (3) - Linux Manual Page.

[93] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[94] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[95] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[96] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0.99*, 2013.

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

[98] Paul Menage. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/cgroups. txt*, 2008.

[99] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[100] Microsoft Corporation. Control Flow Guard (Windows). https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[101] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[102] Mozilla. SSL_CipherPrefSetDefault. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#__SSL_CipherPrefSetDefault_.

[103] National Security Agency. Security-Enhanced Linux. http://selinuxproject.org/.

[104] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[105] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[106] R. NISSIL. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1886.

[107] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[108] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[109] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[110] Ben Niu and Gang Tan. Per-input control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[111] Nokia. Symbian OS Internals - The Loader. http://developer.nokia.com/community/wiki/Symbian_OS_Internals/10._The_Loader#The_loader_server.

[112] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *USENIX Security Symposium*, 2016.

[113] Hilarie Orman. The Morris worm: a fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[114] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security Symposium*, 2013.

[115] PaX Team. PaX address space layout randomization (ASLR). 2003.

[116] PaX Team. PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt, 2004.

[117] Mathias Payer. Too much PIE is bad for performance. 2012. https://nebelwelt.net/publications/12TRpie/gccPIE-TR120614.pdf.

[118] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, 2015.

[119] Mathias Payer and Thomas R. Gross. Fine-grained User-space Security Through Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 157–168, New York, NY, USA, 2011. ACM.

[120] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. Safe Loading - A Foundation for Secure Execution of Untrusted Programs. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 18–32, Washington, DC, USA, 2012. IEEE Computer Society.

[121] Jannik Pewny and Thorsten Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[122] Phrack. Phrack - Volume 0xB, Issue 0x3a. http://phrack.org/issues/58/4.html.

[123] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[124] Google Chromium Project. Undefined Behavior Sanitizer. https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer.

[125] The Wine Project. Wine. https://www.winehq.org/.

[126] Rapid7, Inc. The Metasploit Framework. http://www.metasploit.com/.

[127] RedHat, Inc. Huge Pages and Transparent Huge Pages. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html.

[128] Rich Felker. musl. https://www.musl-libc.org/, 2016.

[129] RISC-V Foundation. riscv-qemu, 2016. https://riscv.org/software-tools/riscv-qemu/.

[130] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to Analyze Binary Computer Code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 798–804. AAAI Press, 2008.

[131] Santa Cruz Operation. System V Application Binary Interface, 2013. http://www.sco.com/developers/gabi/latest/contents.html.

[132] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[133] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*, 2011.

[134] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: a fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[135] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, 2009.

[136] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[137] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security Symposium*, 2001.

[138] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.

[139] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based Static Binary Translator. In *Proc. of the 2012 Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 51–60, New York, NY, USA, 2012. ACM.

[140] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 611–626, Berkeley, CA, USA, 2015. USENIX Association.

[141] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*, 2015.

[142] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[143] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[144] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[145] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.

[146] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. 2016.

[147] Alexandru Sălcianu. Notes on Abstract Interpretation, 2001.

[148] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[149] Quentin Tarantino. The Bonnie Situation. `https://www.youtube.com/watch?v=idV4GQRflHM`.

[150] The Avalanche Project. Avalange - a dynamic defect detection tool. `https://code.google.com/p/avalanche/`.

[151] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014.

[152] Trail of Bits, Inc. MC-Semantics. `http://bit.ly/2geNQEJ`, 2016.

[153] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[154] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. *white paper, Avaya Labs*, 2001.

[155] Ubuntu. Ubuntu Wiki - Security/Features. `https://wiki.ubuntu.com/Security/Features#Built_as_PIE`.

[156] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. PathArmor: Practical ROP Protection Using Context-sensitive CFI. In *ACM CCS*, 2015.

[157] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. PathArmor: Practical ROP Protection Using Context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[158] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2016. IEEE.

[159] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953, 2016.

[160] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele. Accurate Recovery of Functions in a Retargetable Decompiler. In *The 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'12)*, pages 390–392, Amsterdam, NL, 2012. Springer.

[161] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele. Detection and Recovery of Functions and Their Arguments in a Retargetable Decompiler. In *19th Working Conference on Reverse Engineering (WCRE'12)*, pages 51–60, Kingston, Ontario, CA, 2012. IEEE.

[162] Giovanni Vigna. *Static Disassembly and Code Analysis*, pages 19–41. Springer US, Boston, MA, 2007.

[163] Aurélien Wailly. Towards ultimate deobfuscation. Journée Sécurité Lille, February 2015.

[164] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *24th USENIX Security Symp. (USENIX Security 15)*, pages 627–642, Washington, D.C., August 2015. USENIX Association.

[165] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[166] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium*, pages 29–46, 2010.

[167] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[168] Pinghai Yuan, Qingkai Zeng, and Xuhua Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[169] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Defending Virtual Function Tables' Integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[170] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Security and Privacy*, 2013.

[171] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[172] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proc. of the 10th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.

[173] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.

[174] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

[175] ETH Zurich and Universitá di Bologna. PULP, Parallel Ultra Low-Power. `http://www.pulp-platform.org/`.

Appendix A

Symbol Versioning Challenges

In Section 8.3.4 we introduced the concept of symbol versioning, and in Section 8.4.2 we mentioned that its usage introduces additional constraints in the value that `Elf_Rel.r_info` can assume. In this Appendix we illustrate these constraints, and how `leakless` can automatically verify and satisfy them.

A.1 Constraints due to Symbol Versioning

In presence of symbol versioning, the `Elf_Rel.r_info` field is used both as an index into the dynamic symbol table and as an index in the symbol version table (the `.gnu.version` section), which is composed by `Elf_Verneed` values. An `Elf_Verneed` value of zero or one has a special meaning, and stops the processing of the symbol version, which is a desirable situation for the attacker.

To understand the constraints posed by this, we introduce some definitions and naming conventions. idx is the index in `Elf_Rel.r_info` that `leakless` has computed, $\text{baseof}(x)$ is the function returning the base address of section x, $\text{sizeof}(y)$ is the function returning the size in bytes of structure y, and $*$ is the pointer dereference operator. We define the following variables:

$$\begin{aligned} sym &= \text{baseof}(\texttt{.dynsym}) + idx \cdot \text{sizeof}(\texttt{Elf\_Sym}) \\ ver &= \text{baseof}(\texttt{.gnu.version}) + idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) \\ verdef &= \text{baseof}(\texttt{.gnu.version\_r}) + *(ver) \cdot \text{sizeof}(\texttt{Elf\_Verdef}) \end{aligned}$$

To be able to carry on the attack, the following conditions must hold:

1. *sym* points to a memory area controlled by the attacker, and
2. one of the following holds:
   (a) *ver* points to a memory area containing a zero or a one, or
   (b) *ver* points to a memory area controlled by the attacker, which will write a zero value there, or

(c) *verdef* points to a memory area controlled by the attacker, which will place there an appropriately crafted `Elf_Verdef` structure.

All the other options result in an access to an unmapped memory area or the failure of the symbol resolution process, both of which result in program termination.

`leakless` is able to satisfy these constraints automatically in most cases. The typical successful situation results in an *idx* value that points to a version index with value zero or one in the `.text` section (which usually comes after `.gnu.version`) and to a symbol in the `.data` or `.bss` section. A notable exception, where this is impossible to achieve, is in the case of small x86-64 ELF binaries compiled with the support of huge pages [127]. Using huge pages means that memory pages are aligned to boundaries of 2 MiB and, therefore, the segment containing the read-only sections (in particular, `.gnu.version` and `.text`) is quite far from the writeable segment (containing `.bss` and `.data`). This makes it hard to find a good value for *idx*.

A.2 The Huge Page Issue

The effect of huge pages can be seen in the following examples:

```
$ readelf --wide -l elf-without-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x006468 R E 0x1000
  LOAD 0x00407480 0x0005d0 RW  0x1000
  ...

$ readelf --wide -l elf-with-huge-pages

Program Headers:
  Type VirtAddr   MemSiz   Flg Align
  ...
  LOAD 0x00400000 0x00610c R E 0x200000
  LOAD 0x00606e10 0x0005d0 RW  0x200000
  ...
```

While in the first case the distance between the beginning of the executable and the writeable segments is in the order of the kilobytes, with huge pages is more than 2 MiB, and a valid value for *idx* cannot be found.

There are two ways to resolve the problems posed to `leakless` by small 64-bit binaries.

The first option is to find a zero value for `Elf_Verneed` in the read-only segment (usually `.text`). Given *ro_start*, *ro_end* and *ro_size*, as the start and end virtual addresses and the size of the read-only segment respectively, and *rw_start*, *rw_end* and *rw_size* as the respective values for the writeable segment, the following must hold:

$$\begin{aligned} ro\_start \leq\ ver\ &< ro\_end \\ rw\_start \leq sym &< rw\_end \end{aligned}$$

Here, the most difficult case to satisfy is if `.dynsym` or `.gnu.version` start at ro_start. If we assume that *both* hold true, we can write the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_end - ro\_start \\ idx \cdot \text{sizeof}\,(\texttt{Elf\_Sym}) &\geq rw\_start - ro\_start \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &< ro\_size \\ idx \cdot \text{sizeof}\,(\texttt{Elf\_Sym}) &\geq 2 \text{ MiB} \end{aligned}$$

Knowing that `Elf_Verneed` and `Elf_Sym` have, respectively, a size of 2 and 24 bytes for 64 bit ELFs, we can compute the minimum value of `ro_size` to make this system of inequalities satisfiable. The result is 170.7 KiB. If the `.rodata` section is smaller than this size, an alternative method must be used.

The second option is to position `Elf_Verneed` in the writeable segment. In this case, the attack requirements can be described by the following system of inequalities:

$$\begin{aligned} rw\_start \leq\ ver\ &< rw\_end \\ rw\_start \leq sym &< rw\_end \end{aligned}$$

If we, once again, consider the most stringent constraints and apply the previously mentioned assumptions, we get the following:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq rw\_start - ro\_start \\ idx \cdot \text{sizeof}\,(\texttt{Elf\_Sym}) &< rw\_start - ro\_start + \\ &\quad + rw\_size \end{aligned}$$

Or, alternatively:

$$\begin{aligned} idx \cdot \text{sizeof}(\texttt{Elf\_Verneed}) &\geq 2 \text{ MiB} \\ idx \cdot \text{sizeof}\,(\texttt{Elf\_Sym}) &< 2 \text{ MiB} + rw\_size \end{aligned}$$

We can now put a lower bound on the size of the writeable segment (rw_size) to make the system satisfiable: 22 MiB. However, this is unreasonably large, and leads us to the conclusion that this approach is not viable with small 64 bit ELF binaries that use huge pages.

Appendix B

Dal Vangelo secondo LLVM

Fratelli, leggiamo ora un FunctionPass dal vangelo secondo LLVM, file `SROA.cpp`, versetto 18, colonna 72.

In quei tempi, i frontend farisei insozzavano il codice riempiendo le funzioni di `alloca`. La forma SSA era diventata una pura formalita' e variabili promiscue venivano assegnate da piu' statement in ogni dove. `opt` entro' nel tempio dell'intermedio e, fatta una frusta riarrotolando loop dal corpo troppo lungo, inizio' a scacciare le `alloca`, inveendo contro di esse dicendo *dati di poca fede! lasciate che sia la* `regalloc` *a decidere cosa riguarda il Sacro Regno dei registri e cosa verra' condannato a essere gettato nel profondo dello stack!.* Ridotto il suo workload, prima di uscire dal tempio, `opt` disse: *Io sono il figlio di GCC, il quale ci ha fatti a sua immagine ELF e ABI-compatible, ma oggi, io vi porto la buona novella: se rispetterete la forma SSA, vivrete anche dopo la DCE.* Al che, ld.gold, il piu' fedele della toolchain, gli disse: *Branch master, mettimi alla prova, sono pronto a rinnegare il mondo GNU e a seguirti su qualsiasi architettura.* `opt` tacque, e poi emise un warning: *in testing, in testing ti dico: prima che buildbot finisca di compilare la nightly, tu rinnegherai le 3 clausole BSD.* `ld.gold` non seppe come risolvere i simboli generati da `opt`, e si rattristo'. Infatti, egli ancora non sapeva che solo un nuovo shared object poteva riempire i collegamenti mancanti. Si trattava del mistero della trinita' del plugin LTO: il compilatore, l'assembler figlio suo e il linker santo, in una sola entita'.