BARF: A multiplatform open source Binary Analysis and Reverse engineering Framework

Christian Heitman and Iván Arce

Fundación Sadosky,
{cnheitman,iarce}@fundacionsadosky.org.ar

Abstract. The analysis of binary code is a crucial activity in many areas of the computer sciences and software engineering disciplines ranging from software security and program analysis to reverse engineering. Manual binary analysis is a difficult and time-consuming task and there are software tools that seek to automate or assist human analysts. However, most of these tools have several technical and commercial restrictions that limit access and use by a large portion of the academic and practitioner communities. In this paper we introduce BARF, an open source binary analysis framework that aims to support a wide range of binary code analysis tasks that are common in the information security discipline. BARF is a scriptable platform that supports instruction lifting from multiple architectures, binary translation to an intermediate representation, an extensible framework for code analysis plugins and interoperation with external tools such as debuggers, SMT solvers and instrumentation tools. The framework is designed primarily for human-assisted analysis but it can be fully automated.

1 Introduction

1.1 Problem Description

The ability to *inspect* a program and to *understand* how it works is often a strict requirement in many common practices of the information security discipline. Among other things, program analysis is of critical importance to classify programs as benign or malicious, to identify and cluster together semantically equivalent malicious programs in real time attack detection, to discover software security vulnerabilities in benign programs, to determine the set of input data that could trigger a known bug in the program and to assess whether a vulnerability is exploitable.

There is a range of program analysis techniques with a relatively long history of research and development in the computer sciences field, such as model checking, type and effect systems, abstract interpretation, control and data flow analysis and constrain satisfaction problem solving. While these techniques have been regularly applied to security-oriented static and dynamic program analysis with varying degree of success, the vast majority of the existing work is focused on, or relies in the availability of, the source code of the program to be analyzed.

However, today's Information and Communications Technologies (ICT) ecosystem is characterized by a significant dependency on software systems for which the source code is not readily available to parties with a legitimate interest to determine their security properties. As a result, the tools and techniques for security-oriented program analysis that are available and can be applied by practitioners to real-world scenarios is limited.

The state-of-the-art techniques developed in the academic world are usually not directly applicable to binary analysis due to a lack of suitable tools for the purpose or to their lack of effectiveness or efficiency when applied to COTS software components. Generally, publicly available open source binary analysis tools offer a limited set of capabilities and target a small number of architectures (x86, x86-64, ARM) and operating systems (Windows, Linux, OS X).

On the other hand, the relatively low number of commercially available tools that are adequate for multi-architecture, multi-platform binary program analysis, such as IDA [7] and Hopper [8], come with a number of licensing restrictions and price tags that make large scale adoption by information security practitioners an onerous task.

In this paper we present BARF, an extensible open source framework for multi and cross platform binary code analysis, that seeks to provide end to end coverage for all the common program analysis tasks performed by information security practitioners and reverse engineers. The design of BARF encompasses file format recognition and parsing, instruction lifting, binary translation and representation of the program to be analyzed using an intermediate language. The core analysis algorithms are implemented on top of the intermediate language, which makes them architecture- independent, facilitates reuse and reduces the cost of adding support for new architectures. This approach differs from most existing tools, which are tied to specific architectures and operating systems.

1.2 Contributions

In this paper we present BARF, a cross-platform binary analysis framework. Our main contributions can be summarized as follows:

- We provide a design for an extensible platform-independent binary analysis framework. It operates on an intermediate representation of the binary code resulting in a cross-platform framework by design.
- We implement the following core functionalities: a) intermediate language support (REIL) b) an IR emulator, c) integration with a SMT solver and d) translation between intermediate language to SMT expressions. We present a design that allows analysts to model a piece of binary code as formal formulae and to reason about it very easily.
- We give support for the x86 architecture. Support for other architectures can be added easily to the framework as it was designed with that goal.
- We implement two analysis modules that are architecture-independent: a) a module that generates a graph of Basic Blocks of a given binary and b) a code analyzer that resolves constraints on code fragments and checks path satisfiability.

– We built a tool on top of the framework which is a complex example of what can be achieved. It finds ROP gadgets[18] on binary code, classifies them in different types and verifies their semantic.

The framework is open source and can be downloaded from a public repository at `http://github.com/programa-stic/barf-project`.

2 Binary Analysis

The two main approaches in binary analysis are *static binary analysis* and *dynamic binary analysis*. In the first case, the analysis is done without executing the program under analysis. The first step of the analysis involves reading and interpreting the format of the binary program such as ELF, PE, Mach-O, etc. This provides information about the file which includes the location of its different sections, e.g., *data* and *text* sections. Once the text section is located, the disassembly process may begin. This task is not straight forward. Some architectures, for instance, x86, have variable length instructions which makes the process tricky. Moreover, assembly code lacks structure, meaning code and data can be mixed together which complicates the process even further. The last step consists of translation to the intermediate representation language over which analysis algorithms are applied.

In contrast, dynamic analysis requires running the program in order to obtain an execution trace. This is accomplished through dynamic binary instrumentation. There is a lot of information that can be obtained through this process, for instance, the value of each register at each step of the program execution, the list of system calls invoked by a program, etc. Then, the trace is analyzed with offline methods. Limited path coverage is one of the main disadvantages of this technique as only a small subset of all possible paths are exercised during normal execution.

2.1 BARF

The framework was designed with the following goals in mind:

– Cross-platform support : It should be able to analyze binary programs compiled for any available platform (architecture + operating system.)
– Extensibility : It should be simple to add support for new architectures.
– User-oriented : It should provide means for users to create their own analysis tools or adapt existing ones using the framework.

Two of the primary goals of BARF are: a) cross-platform support and b) extensibility, i.e. easy support for new architectures. Both goals are tackled by the use of an intermediate representation language for assembly code. This way, it is possible to abstract architecture-dependent details and work on a common ground. It also encapsulates these details and provides a common interface to the architecture so extensibility to new ones is straightforward.

The criteria to determine the intermediate representation language to use was based on a few desirable properties: a) *expressiveness*, it should be able to model any architecture; b) *side-effect free*, instructions should change program state explicitly (and in just *one* way) and c) *simplicity*, it should be a reduced instruction set so analysis and implementation could be done easily. These are discussed in section 3.1

3 System Overview

Figure 1 depicts the architecture of the framework. It is divided in three main components: Core, Arch and Analysis. The first contains essential modules. The second, provides functionality specific to supported architectures. The last component contains architecture-independent analysis algorithms.

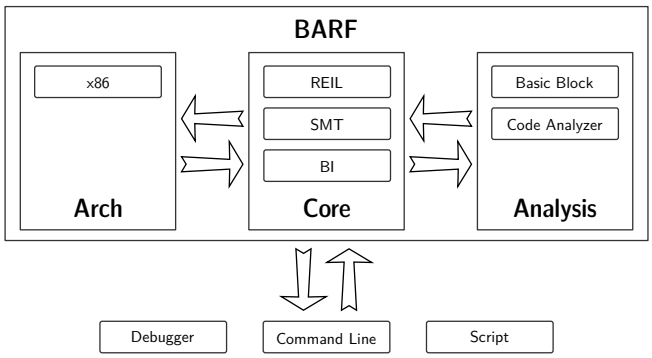

Fig. 1: Architecture of BARF.

The framework relies on external libraries for machine code disassembly and executable format parsing, Capstone [3] and PyBFD [16], respectively.

Core Component It is divided into the following modules: REIL, SMT and BI. These are platform independent and are the building blocks of the rest of the components. The REIL module provides definitions for the REIL language. Also, it implements an emulator and a parser (which is used extensively in instruction translation.) The SMT module consists of two parts. The first one, SMTLIB, a generic interface to SMT solvers (based on PySymEmu [20] SMT support; we currently use Z3 [9] as solver) at the lowest level. It is used to create variables and set assumptions. Also, it can check satisfiability on different sets of formulas. The second part is a module that translates REIL to SMT expressions. Finally, the BI (Binary Interface) module is responsible for loading binary files for processing.

Arch Component Each supported architecture is provided as a subcomponent and is structured in the same way (thus, simplifying framework extension to new

architectures). At the time, we provide support for the Intel x86 architecture (both, 32 and 64 bits).

Each subcomponent provides five well-defined modules. The first describes the architecture, i.e., registers, memory address size and endianness. The second describes the instruction set including information such as number of operands, implicit operands, flags that modifies, etc. The third is the disassembling module which, in this case, is just an interface with the disassembling library. Then comes the parser module, that receives a string with disassembled instructions and produces a series of an annotated objects that describe them. Finally, the translation module provides functionality to transform each assembly instruction into a semantically equivalent sequence of REIL instructions.

Analysis Component This component includes two sample platform-independent modules: Basic Block and Code Analyzer, which will be described in section 4.

3.1 REIL

REIL (Reverse Engineering Intermediate Language) [10] meets the properties discussed in section 2.1. Other IRs were considered, particularly, LLVM IR. We choose REIL, primarily, because of its simplicity. It worth mentioning that existing research on machine code to LLVM IR translation [5,12] was reviewed but the language was judged much more complex than needed for this project.

REIL is a platform-independent intermediate representation of assembly code. Originally designed for static code analysis, it is a low-level language that consists only of 17 instructions. They are grouped in five categories: a) arithmetic, b) bitwise, c) data transfer, d) conditional and c) miscellaneous. Each instruction has exactly three operands, the first two for sources and the third for the destination. There are three type of operands: *Integer Literals*, *Registers* and *REIL addresses*. Registers are string literals, for example, `t0` or `eax`. REIL addresses are composed of two integer parts separated by a period, for example, `4000.05`. The first integer represents the address of the instruction from the source architecture while the second is used as REIL instruction numbering (as one instruction can, and usually is, mapped to more than one REIL instruction.) Some instructions require less than three operand, in that case, a special operand type is used, the *Empty* operand.

As already mentioned, there is a mapping of one to many when translating from the source architecture to REIL as REIL instructions have just one effect on program state. For example, the x86 `add` instruction translation includes the addition operation itself and flags modifications, such as the carry flag (`CF`.)

3.2 SMT

Another aspect of the framework involves interfacing with existing SMT solvers. SMT (Satisfiability Modulo Theories) [9] extends the SAT problem with support for higher level theories, for example, bit-vector arithmetic. These concepts

allows to model semantics of assembly code quite naturally. Consequently, we can reason about code in a simple way.

SMT solvers are very much in use in software security [21] both in static and dynamic analysis. They can be used for vulnerability checking, exploit generation and much more.

REIL can be represented by SMT expression easily as it is a reduced instruction set and each instruction has a single explicit effect on program state. Therefore, translation between both worlds can be accomplished almost directly. Table 1 shows an example of a basic translation. Each register is modeled as a symbol, in this case, a bitvector and memory is modeled as an array of bitvectors.

x86 instruction	REIL instruction	SMT expression
`mov eax, [ebp+0x8]`	`add [DWORD ebp, DWORD 0x8, DWORD t2]`	`(= t2_1 (bvadd ebp_0 #x00000008))`

	`ldm [DWORD t2, EMPTY, DWORD t1]`	`(= (concat`
		`(select MEM (bvadd t2_1 #x00000003))`
		`(select MEM (bvadd t2_1 #x00000002))`
		`(select MEM (bvadd t2_1 #x00000001))`
		`(select MEM (bvadd t2_1 #x00000000)))`
		`t1_1)`

	`str [DWORD t1, EMPTY, DWORD eax]`	`(= t1_1 eax_1)`
`add ebx, eax`	`add [DWORD ebx, DWORD eax, QWORD t3]`	`(= t3_1 (bvadd ebx_0 eax_1))`
	`str [QWORD t3, EMPTY, DWORD ebx]`	`(= ebx_1 t3_1)`

Table 1: REIL to SMT expressions translation example. Note that the translation of `add ebx,eax` includes REIL instructions for the computation of the flags. In this case, they were omitted for the sake of brevity.

4 Analysis Modules

4.1 Basic Blocks

This modules provides functionality for *control flow graph* recovery (CFG.) It operates on the intermediate language (as all modules within this component,) therefore, algorithms implemented here are architecture-independent. It provides two main ways for CFG recovery [22]: *linear sweep* and *recursive descendant*. Additional algorithms can be easily implemented.

4.2 Code Analyzer

This module provides functionality for basic code analysis. It relies on the SMT support provided by the Core component. Instructions can be added to the context of the analyzer, which are taken as assumptions, and then add pre- and post-conditions on variables (memory locations and/or registers) and ask the solver for satisfiability. All the burden of assembly to REIL translation and REIL to SMT expression is taken care of transparently by the framework.

In figure 2 we can see the disassembly code of a function that operates on local variables. In this function there are two memory accesses at addresses

```
80483ed push ebp
80483ee mov ebp, esp
80483f0 sub esp, 0x10
80483f3 mov eax, dword [ebp-0x8]
80483f6 mov edx, dword [ebp-0xc]
80483f9 add eax, edx
80483fb add eax, 0x5
80483fe mov dword [ebp-0x4], eax
8048401 mov eax, dword [ebp-0x4]
8048404 leave
8048405 ret
```

Fig. 2: Assembly code of a function that operates on local variables.

```
80483ed push ebp
80483ee mov ebp, esp
80483f0 sub esp, 0x10
80483f3 mov dword [ebp-0x10], 0x1
80483fa cmp dword [ebp-0xc], 0x41424344
8048401 jne 0x804841c
8048403 cmp dword [ebp-0x8], 0x45464748
804840a jne 0x804841c
804840c cmp dword [ebp-0x4], 0xabcdef
8048413 jne 0x804841c
8048415 mov dword [ebp-0x10], 0x0
804841c mov eax, dword [ebp-0x10]
804841f leave
8048420 ret
```

Fig. 3: Assembly code of a function with multiple branches.

`ebp-0x8` and `ebp-0xc`, respectively (lines 4-5.) These values are then used in some calculations (two successive additions, lines 6-7) and the result is stored in memory (line 8) and returned in the `eax` register (line 9). One possible question that arises is what should be the values of the local variables so the function returns a specific (desired) value. Figure 4 shows how to do exactly this. First, it loads the instruction to be analyzed (lines 2-5). Then, it models registers and memory locations involved in the operation (lines 8-12) and establishes conditions that should hold (lines 15-19). Finally, it checks for satisfiability (line 22). If all conditions are satisfiable, it asks for possible values for the registers and memory (lines 24-27). In this example, the constraint set is satisfiable and one possible solution is: `a = 3` and `b = 5`.

```
# Add instructions to analyze
translation = barf.translate(0x80483ed, 0x8048401)
for addr, asm_instr, reil_instrs in translation:
  for reil_instr in reil_instrs:
    barf.c_analyzer.add_instruction(reil_instr)

# Get SMT expression for registers and memory
eax = barf.c_analyzer.get_reg_expr("eax")
ebp = barf.c_analyzer.get_reg_expr("ebp")
a = barf.c_analyzer.get_mem_expr(ebp-0x8, 4)
b = barf.c_analyzer.get_mem_expr(ebp-0xc, 4)
c = barf.c_analyzer.get_mem_expr(ebp-0x4, 4)

# Set range for variable a and b
barf.c_analyzer.set_precondition(a>=2, a<=100)
barf.c_analyzer.set_precondition(b>=2, b<=100)

# Set desired value for the result
barf.c_analyzer.set_postcondition(c==13)

# Check satisfiability
if barf.c_analyzer.check() == "sat":
  # Get concrete value for expressions
  print barf.c_analyzer.get_expr_value(eax)
  print barf.c_analyzer.get_expr_value(a)
  print barf.c_analyzer.get_expr_value(b)
  print barf.c_analyzer.get_expr_value(c)
```

Fig. 4: Extract of an analysis script for assembly code on figure 2.

```
# Recover control flow graph
cfg = barf.recover_cfg(0x80483ed, 0x8048420)

# Get SMT expression for registers and memory
esp = barf.c_analyzer.get_reg_expr("esp")
ebp = barf.c_analyzer.get_reg_expr("ebp")
rv = barf.c_analyzer.get_mem_expr(ebp-0x10, 4)
a = barf.c_analyzer.get_mem_expr(ebp-0xc, 4)
b = barf.c_analyzer.get_mem_expr(ebp-0x8, 4)
c = barf.c_analyzer.get_mem_expr(ebp-0x4, 4)

# Set stack pointer
barf.c_analyzer.set_precondition(esp==0xbfffceec)

# Traverse paths and check satisfiability
paths = cfg.all_simple_bb_paths(0x80483ed, 0x8048420)

for path in paths:
  # If it is satisfiable, get possible assignments
  # for memory and registers
  if barf.c_analyzer.check_path_sat(path, 0x80483ed):
    print barf.c_analyzer.get_expr_value(rv)
    print barf.c_analyzer.get_expr_value(a)
    print barf.c_analyzer.get_expr_value(b)
    print barf.c_analyzer.get_expr_value(c)
```

Fig. 5: Extract of an analysis script for assembly code on figure 3.

Another assembly code is shown in figure 3. This code has multiple branches (lines 6, 8, 10) which means there are multiple paths from the function entry to the exit node. The framework implements a function to list all paths between two basic blocks and the code analyzer provides functionality to know which of all these paths are feasible as well as possible values assignments for the registers and variables involved. Figure 5 shows an extract of a script that does this checking. First, it recovers the control flow graph of the function (line 2). Then, as in the previous example, it models the registers and memory locations involved as SMT expressions (lines 5-10). Finally, it iterates all simple path (path with no loops) between the entry and exit basic blocks (line 18). For each path (list of basic blocks) it check satisfiability and if so, it gets possible assignments for registers and variables (lines 22-25). The path satisfiability check burden is encapsulated within the `CodeAnalyzer` class. It basically adds each instruction from each basic block in the path to the analyzer and includes restrictions on the jump conditions forcing them to be true or false according to the case.

5 Case Study

Return-oriented programming (ROP) is a technique for software exploitation which allows arbitrary computation by an attacker through the use of code fragments –called *gadgets*– "borrowed" from the attacked program without the need of code injection [24]. This technique is necessary to achieve successful exploitation on modern operating systems as most of them implement mechanisms of data execution prevention.

5.1 ROP Gadgets

In addition to the analysis modules implemented and described in the previous section, we used BARF to create a tool to *automatically* search, classify and verify ROP gadgets [18] in x86 binary programs. The tool is based on the ideas exposed by Schwartz et al. in [17] and shows some of the capabilities of the framework. It is a step forward in the direction of tools for automatic exploit generation [17,1]. The tool has three stages. In the *search* stage, it looks for `ret` instructions in the whole text section of a given binary. Once all `ret` instructions have been identified, it disassembles backwards from these addresses trying to recover valid instructions (this is done according to the algorithm described in [18].) Sequences of instructions that disassembled correctly ending with a `ret` instruction are considered candidate gadgets. The second stage, *classification*, goes through the list of candidate gadgets and classifies each one of them in different types (arithmetic, memory read/write, etc) through emulation. This means that each gadget is executed using the REIL emulator provided by BARF and its output analyzed. For instance, if we found that a register contains the sum of other two (previously assigned with random values), then, we classify that gadget as an arithmetic gadget. The third stage, *verification*, involves the use of the SMT solver to verified that the type assigned in the previous step

is the correct one. It is worth noting that the last two stages are architecture-independent. So, in principle, they can be applied to any supported architecture provided that a list of gadget is given. The only architecture dependent part is the first stage, the one involved in searching for gadgets. This happens due to some architecture-specific details that have impact on gadget discovery. For instance, in the x86 architecture one can search for `ret`-ended gadgets through all the text section of a binary program with byte granularity. On other architectures, e.g. SPARC, this cannot be done as instructions are aligned and gadgets have to be searched in a different way. However, a general search stage can be accomplished considering the details mentioned in [11] since disassembly and translation of instructions to REIL is done transparently.

6 Related Work

There are several binary analysis frameworks. All of them differ in the approach – they were motivated by different applications– as well as the set of functionalities they provide.

BitBlaze [19] is a project that aims to provide a full binary analysis stack covering almost the whole spectrum of related tasks. It is composed of: a) Vine, a static analyzer; b) TEMU, a dynamic analyzer and c) Rudder, a symbolic execution engine. BAP [2] is a another framework based on Vine. It operates on an intermediate language called BIL which explicitly represents all side effects of assembly instructions. BAP has several built-in analyses and can perform symbolic execution. Insight [13] is a static analysis framework designed for Unix-like systems, supports i386 and amd64 architectures and can load many file formats, i.e., PE, ELF, Mach-0. It has its own IR called Microcode and provides tools to manipulate and transform binary code. Jakstab [14] is another static analysis framework based on abstract interpretation and designed to support multiple architectures using customized instruction decoding (currently, supports x86 (32 bits) for PE and ELF file formats.) It uses an IR inspired by the semantics specification language (SSL) [6]. Analysis algorithms run on top of this language and translation is done on the fly as the analysis is performed. Finally, the radare project [23] is built under the Unix-liked concepts such as "everything is a file" and "small programs that interact through standard I/O." Currently is composed of a set of utilities including a debugger, an assembler/disassembler and a binary diffing tool.

7 Conclusion & Future Work

In this paper, we presented BARF, a binary analysis framework that aims at being a cross-platform analysis tool. BARF provides means for analysis and verification on binary code. Its intermediate language, based on REIL, encode explicitly the side effects of the instructions of the source architecture. We, also, presented a tool for ROP gadget finding built upon the framework. Our next step is to provide support for the ARM architecture. Also, we plan to give support for

symbolic execution and taint analysis to cover a broader spectrum of common binary analysis requirements.

References

1. Avgerinos, Thanassis, et al. "AEG: Automatic Exploit Generation." NDSS. Vol. 11. 2011.
2. Brumley, David, et al. "BAP: A binary analysis platform." Computer Aided Verification. Springer Berlin Heidelberg, 2011.
3. Capstone, `http://www.capstone-engine.org/`
4. Cha, Sang Kil, et al. "Unleashing mayhem on binary code." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
5. Chipounov, Vitaly, and George Candea. "Dynamically Translating x86 to LLVM using QEMU". No. EPFL-REPORT-149975. 2010.
6. Cifuentes, Cristina, and Shane Sendall. "Specifying the semantics of machine instructions." Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on. IEEE, 1998.
7. IDA, Interactive Disassembler `https://www.hex-rays.com/products/ida/`
8. Hopper, OS X and Linux disassembler `http://www.hopperapp.com/`
9. De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008. 337-340.
10. Dullien, Thomas, and Sebastian Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis." Proceeding of CanSecWest (2009).
11. Dullien, Thomas, Tim Kornau, and Ralf-Philipp Weinmann. "A Framework for Automated Architecture-Independent Gadget Search." WOOT. 2010.
12. Fracture `https://github.com/draperlaboratory/fracture`
13. Insight, `https://insight.labri.fr/trac`
14. Kinder, Johannes, and Helmut Veith. "Jakstab: A static analysis platform for binaries." Computer Aided Verification. Springer Berlin Heidelberg, 2008.
15. Li, Lixin, and Chao Wang. "Dynamic analysis and debugging of binary code for security applications." Runtime Verification. Springer Berlin Heidelberg, 2013.
16. PyBFD, `https://github.com/Groundworkstech/pybfd`
17. Schwartz, Edward J., Thanassis Avgerinos, and David Brumley. "Q: Exploit Hardening Made Easy." USENIX Security Symposium. 2011.
18. Shacham, Hovav. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)." Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007.
19. Song, Dawn, et al. "BitBlaze: A new approach to computer security via binary analysis." Information systems security. Springer Berlin Heidelberg, 2008. 1-25.
20. PySymEmu, `https://github.com/feliam/pysymemu`
21. Vanegue, Julien, Sean Heelan, and Rolf Rolles. "SMT Solvers in Software Security." WOOT. 2012.
22. Linn, Cullen, and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly." Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003.
23. radare, `http://radare.org`
24. Bratus, Sergey, et al. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." USENIX; login (2011): 13-21.