Insight: An Open Binary Analysis Framework

Emmanuel Fleury[1], Olivier Ly[1], Gérald Point[2], and Aymeric Vincent[3]

LaBRI, UMR 5800, Talence, France
[1]Université de Bordeaux, Talence, France
[2]CNRS, Talence, France
[3]INP Bordeaux Aquitaine, Talence, France
{emmanuel.fleury,olivier.ly,gerald.point,aymeric.vincent}@labri.fr

Abstract. We present Insight, a framework for binary program analysis and two tools provided with it: CFGRecovery and `iii`.

Insight is intended to be a full environment for analyzing, interacting and verifying executable programs. Insight is able to translate `x86`, `x86-64` and `msp430` binary code to our intermediate representation and execute it symbolically in an abstract domain where each variable (register, memory cell) is substituted by a formula representing all its possible values along the current execution path.

CFGRecovery aims at automatically rebuilding the program control flow based only on the executable file. It heavily relies on SMT solvers.

`iii` provides an interactive and a (Python) programmable interface to a coherent set of features from the Insight framework. It behaves like a debugger except that the execution traces that are examined are symbolic and cover a collection of possible concrete executions at once. For example, `iii` allows to perform an interactive reconstruction of the CFG.

Keywords: binary analysis, CFG recovery, symbolic debugging.

1 Introduction

Nowadays, finding complex bugs automatically has become fruitful and useful. Yet, most of software analysis techniques rely on the fact that a complete blueprint of the program is available (full specifications, formal design documents, source code) at a level of abstraction suitable for analysis.

A recent interest has been shown in analyzing executable programs with no prior knowledge of their internals [11,2,4]. These efforts have been essentially pushed forward by the need to get some trust on external binary-only software, or analyzing potentially malicious software.

But, one of the main problems of binary analysis is to rebuild a correct control flow graph of the program which can be made difficult to recover because of data-entanglement, self-modifying code, or other binary specific effects (intentional or not) linked to this specific format. It is needed because most, if not all, the analysis techniques require the control flow graph to operate, which means that the recovery of the control flow comes before any other usual analysis. Moreover,

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 218–224, 2015.
DOI: 10.1007/978-3-662-46681-0_18

depending on the completeness and the accuracy of the recovery, the analysis may succeed or fail. Thus, in order to leverage existing techniques on higher-level code, the first step will be to recover the control flow as accurately as possible.

A few pioneers of binary analysis already made advances on recovery techniques [7,12]. But, recent works [11,2,4] led to new approaches for both recovery and/or analysis on binary programs and the design of new tools: McVeto [14], CodeSurfer/x86 [1], OSMOSE [4], Jakstab [10], or frameworks: BitBlaze [13], BAP [6], Otawa [3]. Yet, few of these tools are actually open platforms which could be used by the community to ease the cumbersome steps of working on binary programs for new researchers in this field.

INSIGHT is a framework, including a library and tools, aiming to provide an environment to perform binary analysis for verification purposes. Yet, even if our first intent was binary verification, one may use the framework for other goals such as program control flow extraction, reverse engineering, decompilation, ... As a first step, we built a complete chain of modules that can be used to extract concrete or symbolic traces of the binary program in a simulated environment, translate it into our intermediate representation and perform analyses on it.

These modules have been combined into two tools that are now part of the framework: CFGRECOVERY, an automated tool to recover the control flow of the binary program and the INSIGHT Interactive Inspector (`iii`), an interactive tool working as a debugger to execute and interact with both the original binary program and its intermediate representation.

INSIGHT was started during the BINCOA ANR-funded project [5]. The framework is not currently focused on performance, and on small programs of a few kilobytes, a couple of minutes of computation on an Intel Core i5 laptop are to be expected to recover the control flow graph using Mathsat [8] as the SMT solver. Starting stepping through a program of any size under `iii` is immediate. INSIGHT, CFGRECOVERY and `iii` are freely available with an open source 2-clause BSD license (visit `http://insight.labri.fr`).

In the following we first present the library, then the tools, and we conclude with future research directions.

2 The Insight Library

The INSIGHT library gathers all the functions, data structures and algorithms that allow to build tools for binary analysis. It includes primitives to handle our intermediate representation, which is called the *microcode*, the functions used to extract and translate the original assembly into microcode and a way to execute it within a simulated environment on a given abstract domain.

2.1 Insight's Microcode

Binary instructions are translated into an architecture-independent representation called *microcode*. Fig. 1 gives an example of *microcode*. The *microcode* is an oriented graph whose nodes are labelled with addresses (*e.g.* `[0x0,0]`) and edges with a guard (`≪...≫`) and an instruction (*e.g.* an assignment `var:=expr`).

The addresses which label nodes are composed of two parts: a global address which corresponds to the address as seen by the binary program, and a local sub-address which allows to translate one assembly instruction into a sub-graph. The guards are formulae with a boolean value, and the edge can exist in the semantics of the microcode in a given context only if the guard evaluates to true in that context.

There are three types of instructions: a ***skip*** instruction which does nothing but go to its successor (*e.g.* at `[0x1,0]` a conditional jump); all static jumps are implemented with a skip instruction. An ***assignment*** which assigns the value of an expression to a l-value (*e.g.* `eax` is assigned to at `[0x5,0]`). And, a ***dynamic jump*** which has no successor in the graph but provides an expression determining the global address where execution should continue (*eg.* at `[0x3,0]` a jump to the value of `eax`).

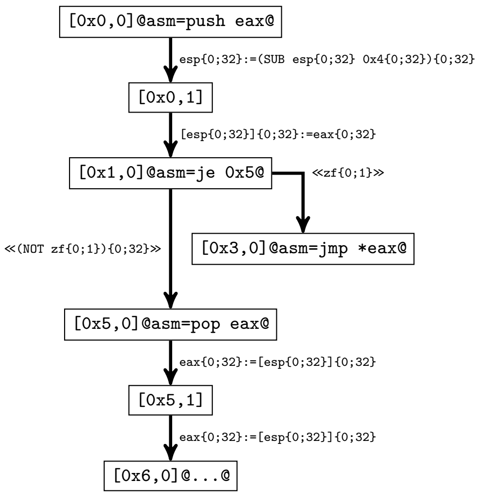


Fig. 1. Microcode example (from x86 asm)

Expressions can use a variety of bitvector operators (addition, bits extraction, ...), and base operands are made of constants, variables, and memory references. Every subexpression includes the possibility of extracting a bitvector. This way, subbitvectors of variables and memory references constitute acceptable l-values and are legitimate expressions.

2.2 Microcode Providers and Handling

One of the very appealing features of Insight is its ability to load a binary program and translate it into microcode. This feature is provided thanks to GNU's `libbfd` which allows to open almost any executable container format (*e.g.* ELF, PE-COFF). Translation from binary assembly instructions into microcode is provided by Insight itself, but uses the GNU `libopcodes` as a first step. This translation is currently implemented almost fully for 32-bit x86, 64-bit x86 and 16-bit MSP430. Yet, it is important to notice that only integer datatypes are supported (no floating point, SIMD, ...) as is the case of the other binary analysis software that we know of.

A handful of classes are used to represent a microcode program. In order to ease creation of microcode and thus the writing of decoders, a very simple API is provided to add microcode instructions to a program. Furthermore, a very useful feature is the ability to annotate almost any object of microcode. For example, a microcode node corresponding to a given address can be annotated by the textual representation of the assembly instruction at that address; dynamic jumps can be annotated by their potential targets; and so on. This gives a homogeneous place for analyses to store their results and helps provide the end-user with information related to a given microcode part.

2.3 Simulation on Domains

Mainly, two domains are provided: a "*concrete*" domain which allows computations of a single value per l-value and provides the usual operations on bitvectors. And, a "*symbolic*" domain which represents sets of values thanks to assertions constraining variables and memory elements. Two additional toy domains are also provided: the "*sets*" domain which uses sets of concrete values to represent possible values, and the "*intervals*" domain which uses a pair of integers to represent an interval of concrete values.

The simulation on the symbolic domain is the one we massively rely on for recovering the control flow of the program. Indeed, we use symbolic execution to collect program traces and build a microcode program from it. This technique has already been used for many other purposes like automatic software testing [4] or processor microcode verification [9].

More precisely, *symbolic execution* is performed by the simulation engine that will execute every step of the program assuming symbolic values for inputs rather than concrete ones. Our symbolic domain is the set of all the (quantifier free) bitvector arithmetic formulae, which allows to represent exactly the semantics of assembly instructions.

3 CFGRecovery

CFGRECOVERY is a tool dedicated to the recovery of the control flow of an executable program in the most accurate way, only based on the binary form of the program. Several classical disassembly strategies may be chosen (*linear sweep*, *recursive traversal*). But, our main disassembly method is to use an under-approximation strategy using *symbolic execution* in order to avoid spurious execution traces and to output a possibly incomplete but trusted control flow graph. A very simple example is given in Fig. 2, it shows the disassembly of code with instruction overlapping obfuscation using `objdump` on the left, and with CFGRECOVERY on the right. Note that CFGRECOVERY is accurate.

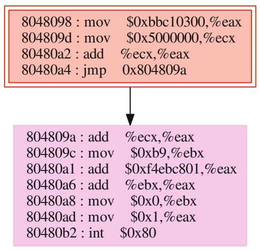
4 Insight's Interactive Inspector (`iii`)

INSIGHT's interactive inspector (`iii`) is a cross-debugger using an abstract domain to represent memory and register values. The `iii` tool is a Python interpreter enriched with INSIGHT library features. As for CFGRECOVERY, it can

```
instruction_overlapping-i386: file format elf32-i386
Disassembly of section .text:

08048098 <_start>:
 8048098:  b8 00 03 c1 bb  mov  $0xbbc10300,%eax
 804809d:  b9 00 00 00 05  mov  $0x5000000,%ecx
 80480a2:  01 c8           add  %ecx,%eax
 80480a4:  eb f4           jmp  804809a <_start+0x2>
 80480a6:  01 d8           add  %ebx,%eax
 80480a8:  bb 00 00 00 00  mov  $0x0,%ebx
 80480ad:  b8 01 00 00 00  mov  $0x1,%eax
 80480b2:  cd 80           int  $0x80
```

1a. objdump (linear sweep) disassembly.



```
8048098 : mov   $0xbbc10300,%eax
804809d : mov   $0x5000000,%ecx
80480a2 : add   %ecx,%eax
80480a4 : jmp   0x804809a
```

```
804809a : add   %ecx,%eax
804809c : mov   $0xb9,%ebx
80480a1 : add   $0xf4ebc801,%eax
80480a6 : add   %ebx,%eax
80480a8 : mov   $0x0,%ebx
80480ad : mov   $0x1,%eax
80480b2 : int   $0x80
```

1b. CFGRecovery disassembly.

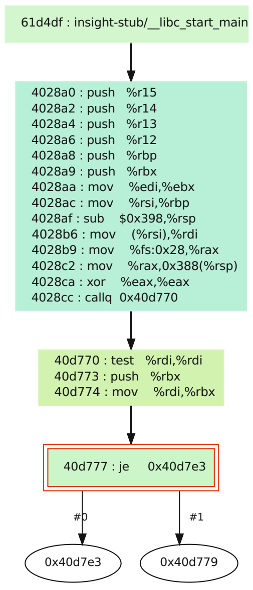
Fig. 2. Example of code disassembled by objdump (1a) and CFGRecovery (1b)

load binary executable files and simulate them over any domain supported by the framework.

The basic principle of operation of iii is that a microcode program is continuously maintained in memory and is enriched by explicit loading of microcode, or by exploring a binary executable using symbolic execution. At each step, an edge of the microcode is followed, and any location which is encountered and not yet part of the microcode will be added to it.

Many usual debugger commands are available in iii, possibly adapted to its specificities. For example, the step() function follows the microcode edges associated with a full assembly instruction, but also the microstep() function follows just one edge of the microcode. Another example is the cont() function which will continue until one of the usual conditions occurs (breakpoint or "end of program") or when non-determinism is encountered, in which case the user is asked to select which edge to follow.

It is also possible to load microcode stubs at any address in the code prior to reaching that address. We usually use stubs to replace a call to an external procedure by a simplified model of it. These stubs can be loaded at a given address and "folded" into this global address by letting the stub loader replace all other global addresses by local addresses. This allows to preserve global address space whose usage is dictated by the binary program.



```
61d4df : insight-stub/__libc_start_main
```

```
4028a0 : push   %r15
4028a2 : push   %r14
4028a4 : push   %r13
4028a6 : push   %r12
4028a8 : push   %rbp
4028a9 : push   %rbx
4028aa : mov    %edi,%ebx
4028ac : mov    %rsi,%rbp
4028af : sub    $0x398,%rsp
4028b6 : mov    (%rsi),%rdi
4028b9 : mov    %fs:0x28,%rax
4028c2 : mov    %rax,0x388(%rsp)
4028ca : xor    %eax,%eax
4028cc : callq  0x40d770
```

```
40d770 : test   %rdi,%rdi
40d773 : push   %rbx
40d774 : mov    %rdi,%rbx
```

```
40d777 : je     0x40d7e3
```

Fig. 3. iii CFG exploration

At any moment, the (symbolic) content of memory and registers can be displayed similarly to what can be done in a debugger. The current microcode can also be displayed graphically with the current microcode node in the simulation trace highlighted, and hotkeys allow to call functions like step() to extend the trace from within the graph. See Fig. 3.

5 Future Directions

INSIGHT has now reached a level of achievement that allows to extract a coherent microcode model from possibly complex software and interact with it. Many new ideas can be explored with the framework and we hope the community will take advantage of this massive open source code base. We intend to further research on topics such as self-modifying code, loop summarization, and verification. Also practical usage of the framework for reverse engineering purposes is a promising lead.

Acknowledgments. We would like to thank all the contributors, and especially R. Tabary for his work on INSIGHT and also for the `crackme` example used in the demo.

References

1. Balakrishnan, G., Gruian, R., Reps, T., Teitelbaum, T.: CodeSurfer/x86—A platform for analyzing x86 executables. In: Bodik, R. (ed.) CC 2005. LNCS, vol. 3443, pp. 250–254. Springer, Heidelberg (2005)
2. Balakrishnan, G., Reps, T.: WYSINWYX: What You See Is Not What You eXecute. Journal of ACM Transactions on Programming Languages and Systems (TOPLAS) 32 (2010)
3. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An open toolbox for adaptive WCET analysis. In: Min, S.L., Pettit, R., Puschner, P., Ungerer, T. (eds.) SEUS 2010. LNCS, vol. 6399, pp. 35–46. Springer, Heidelberg (2010)
4. Bardin, S., Herrmann, P.: OSMOSE: automatic structural testing of executables. Software Testing, Verification and Reliability 21(1), 29–54 (2011)
5. Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A.: The BINCOA framework for binary code analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 165–170. Springer, Heidelberg (2011)
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 463–469. Springer, Heidelberg (2011)
7. Cifuentes, C.: Reverse Compilation Techniques. Ph.D. thesis, Queensland University of Technology, Department of Computer Science (1994)
8. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
9. Franzn, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: Proc. of Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2010), pp. 121–128. IEEE (2010)
10. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
11. Kinder, J., Zuleger, F., Veith, H.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)

12. Mycroft, A.: Type-based decompilation (or program reconstruction via type reconstruction). In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 208–223. Springer, Heidelberg (1999)
13. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Zhenkai, K.M.G.a.L., James, N., Pongsin, P., Prateek, S.: BitBlaze: A new approach to computer security via binary analysis. In: Proc. of Int. Conf. on Information Systems Security (ICISS). LNCS, pp. 1–25. Springer, Heidelberg (2008)
14. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)