Binsec platform overview

Adel Djoudi and Sébastien Bardin

January 20, 2015

Contents

1 Disassembly methods

The BinSec platform implements four basic disassembly methods:

- **Linear sweep.** In this method, instructions are decoded sequentially starting from a list of initial addresses and stepping to the next address according to the current instruction size. Note that the user can choose between instruction-wise disassembly or byte-wise disassembly. This method is used by programs such as the GNU utility `objdump`, and can be activated in our tool by the option `-linear` in disassembly mode.
- **Recursive traversal.** This method follows the control flow of the program, but it cannot follow successors of dynamic jumps. Yet, we allow the user to specify potential jump targets at each dynamic jump. This method can be activated in our tool by the option `-rec` in disassembly mode.
- **Linear sweep combined with recursive traversal.** This method merges the two previous ones. It is similar in some way to the method used by `IDA Pro`, and can be activated in our tool by the option `-rec-linear` in disassembly mode.
- **Dynamic disassembly.** In this mode, a specified numer of random executions are launched in order to detect function entry points and jump targets, then recursive traversal disassembly is performed with this additional information.

The info file allows to provide information to disassmbly methods listed above. The directive `@recursive disassembly` allows to specify a worklist of initial addresses to start recursive disasssembly from, and the directive `@linear disassembly` allows to specify ranges of addresses for linear disassembly. An example is shown in Figure 1 .

```
@recursive disassembly :
  0x0804810d; 0x0804809c;

@linear disassembly:
  (0x0804810d, 0x08048167) (0x080480d8, 0x080480fc)
```

Figure 1: info file: disassembly directives

2 Simplifications

Simplifications are inspired by standard code optimization techniques used in compilers, especially constant propagation and liveness analysis. Yet, we design our techniques such that they remain sound in case of incomplete CFG, which is a common case in binary-code analysis.

The goal of our simplification mechanism is to lighten some undully heavy DBA translations, we do not seek to optimize the original program. Having this point in mind, we focus on removing as much as possible assignments to flags ("flag assignments") since they are very likely to be useless. Assignments to temporary variables ("temporary assignments") are also a target of choice. The impact of our simplifications is shown in Figure 2.

```
# 31 ed    xor ebp, ebp
res32 := 0<32>;
OF := 0<1>;
SF := res32{31,31};
ZF := (res32 = (cst, 0<32>));
CF := 0<1>;
ebp := 0<32>;

#0f 85 7c 00 00 00  jnz 0x805637c
if (!ZF) goto L1
else goto L2

L1: #83 e3 fa    and ebx, 0xfffffffa
ebx := (ebx and 0xfffffffa<32>);
OF := 0<1>;
SF := res32{31,31};
ZF := (res32 = 0<32>);
CF := 0<1>;

L2: #85 ff    test edi, edi
res32 := edi;
OF := 0<1>;
SF := res32{31,31};
ZF := (res32 = 0<32>);
CF := 0<1>;
```

Before simplification

```
# 31 ed    xor ebp, ebp



ZF := 1<1>;

ebp := 0<32>;

#0f 85 7c 00 00 00 jnz  0x805637c
if (!ZF) goto L1
else goto L2

L1: #83 e3 fa  and ebx, 0xfffffffa
ebx := (ebx and 0xfffffffa<32>);









L2: #85 ff test  edi, edi
res32 := edi;
OF := 0<1>;
SF := res32{31,31};
ZF := (res32 = 0<32>);
CF := 0<1>;
```

After simplification

Figure 2: Example of simplifications

Simplification layers. Our simplification method is organized around three layers:

- instruction simplification: idiom expressions can be turned into constant values through rewriting rules. Such expression are largely met in machine code, For instance, `eax xor eax` can be turned into 0.
- intra-block simplification: this layer performs constant propagation inside a DBA block, liveness analysis on temporary variables (recall that by definition they are killed at the end of the block) and other forms of temporary variable eliminations.
- inter-block simplification: this layer performs liveness analysis on flag variables, we remove flag variables which are sure to be killed (the usual ¬may-used approach from compilers being unsound in case of incomplete CFGs).

Simplification levels. Simplifications can be performed on the whole program at once, *per* function or *per* sequence (i.e. *per* CFG block). These three

levels of simplifications give different trade-offs between computation time and quality of the simplification. We found that the function-level approach offers large simplifications in a reasonable amount of time.

3 Memory model and Simulation

This service allows to evaluates DBA intermediate representation. For instance, the evaluator can be used to perform randomized testing or check the consistency between the DBA program semantics and the real program's semantics. The evaluation can be performed in three distinct memory models (flat, region-based and low-level region-based).

Low-level region based memory model. The Region-Based memory model presents some limitations when dealing with low level operations. Illegal operations are for example:

- $(r_1, v_1) + (r_2, v_2) = \bot_V$ if $r_1, r_2 \neq Cst$
- $(r_1, v_1) - (r_2, v_2) = \bot_V$ if $r_1 \neq r_2$

Yet, such patterns are found in libc programs, such as `memmove` or `memcopy`, and can also be introduced at compile-time (branchless conditions). We use symbolic values to keep an intermediate representation for the evaluated expressions. A concrete value is retrieved from symbolic values as soon as possible through a dedicated rewriting engine. Actually, our implementation supports three different memory models:

1. Flat memory model (if we consider only the Cst region)
2. Region-based memory model (without symbolic values)
3. Low-level region-based memory model (with symbolic values)

model	simulation	scalability of analysis
flat	✓	×
region	×	✓
low-level region	✓	✓

Figure 3: Comparison of standard memory models

A DBA platform : command line options

Command	Argument		Description
disas	-dmode	rec	(Default) Activates recursive disassembly mode. A work-list of initial disassembly addresses is retrieved from the info file.
		linear	Activates linear disassembly mode. A list of ranges of addresses is retrieved from the info file.
		reclinear	Activates linear disassembly mixed with recursive disassembly mode.
		bytelinear	Activates linear bitwise disassembly mode.
	-out	[`file_name`]	Indicates the output file where to display DBA instructions. By default, a file named `out.dba` is created in current directory.
	-outop	[`file_name`]	Indicates the output file where to display opcodes, if this option is not specified the opcodes are displayed on standard output.
simulate	-dba		(Mandatory) Indicates the dba file.
	-start	[`hex_address`]	(Mandatory) Indicates the initial address of the program.
	-fuzz	[`positive_integer`]	Indicates the number of fuzzing iterations.
	-step		Activates step (instruction) by step (instruction) simulation.
	-mem-mode	flat	Activates simulation with flat memory model.
		region	Disables the use of symbolic values. Activates simulation with pure region-based memory model. By default, the low-level region-based memory model with mixed basic and full symbolic values is used.
		rewrite	Disables full symbolic values but enables basic symbolic values. Simulation still in low-level region-based memory model.
		logic	Disables basic symbolic values but enables full symbolic values. Simulation still in low-level region-based memory model.
	-else-default		Simulation follows else branches if simplification fails on symbolic conditions.
	-vv		Verbose display of registers and memory content after simulation.
	-v		Verbose display of registers only after simulation.
analyse	-dba		(Mandatory) Indicates the dba file.
	-start	[`hex_address`]	(Mandatory) Indicates the initial address of the program.
	-kmax		Indicates the maximum bound of the ksets cardinality.
	-clos		Disables disassembly during the analysis. The analysis is restricted to the content of DBA file specified by -dba option.
	-degrade		Allows analysis to switch to unsound mode whenever stumbling on (jump $\top$) by propagating the last non-$\top$ computed approximation.
	-vv		Verbose display of registers and memory content after analysis.
	-v		Verbose display of registers only after analysis.
*	-info	[`file_name`]	Indicates the info file name, if this option is not specified a `in.info` file is sought by default in current directory.
	-loader	[`file_name`]	Specifies the loader library. Currently only the elf file format is supported. If this option is omitted, the executable file is converted into a simple table of bytes with indexes starting from 0.
	-file	[`file_name`]	Indicates the executable file.
	-simplify-level	prog	Enables DBA simplifications on whole disassembled instructions at once.
		fun	Enables DBA simplifications per function.
		seq	Enables DBA simplifications per Sequence.

* : disas, simulate, analyse