BINSEC: Binary Code Analysis with Low-Level Regions[⋆]

Adel Djoudi and Sébastien Bardin

CEA, LIST, Gif-sur-Yvette, F-91191, France
first.name@cea.fr

Abstract. This article presents the open source BINSEC platform for (formal) binary-level code analysis. The platform is based on an extension of the DBA Intermediate Representation, and it is composed of three main modules: a front-end including several syntactic disassembly algorithms and heavy simplification of the resulting IR, a simulator supporting the recent low-level region-based memory model, and a generic static analysis module.

1 Introduction

Binary-level program analysis has gained interest in these last years in order to address the problems of analyzing closed-source software or mobile code (including malware) and detecting compiler-induced bugs. Not requiring source code makes such analysis widely applicable.

The goal of BINSEC is to ease the development of binary code analyzers by providing an open formal model for binary programs and an open-source platform allowing to share front-ends and ISA support. Like other platforms such as BAP [7], GDSL [12], Jakstab [11] or OSMOSE [3, 4], our platform disassembles binary code and translates the resulting machine instructions into an intermediate language, which is then analyzed. The main novelties of BINSEC are the following:

- an extended Intermediate Representation (IR) providing abstraction and specification mechanisms (Section 2), contrary to the very operational nature of previous proposals [5, 7, 9, 13];
- a low-level region-based semantics [2], allowing both an abstract view of the memory and the ability to simulate correctly many native codes (Section 3.3);
- a simplification engine able to remove a large part of flag operations (Section 3.2).

BINSEC is open-source (lgpl), it is written in OCaml and it is available at
`http://sebastien.bardin.free.fr/binsec/`.

2 Intermediate Representation: Extended DBA

DBA model. Dynamic Bit-vector Automata (DBA) [5] have been proposed as a generic and concise formal model for low-level programs. They offer the following advantages:

⋆ Work partially funded by French ANR (project BINSEC, grant ANR-12-INSE-0002).

(1) an architecture-independent formalism, (2) a very concise set of instructions and operators, and (3) a simple semantics, without any implicit side-effect. They have been used for modeling PowerPC and a few other architectures in previous binary-level analyzers [3, 4, 6]. Note that floating-point arithmetic, multi-thread and self-modification are currently outside of the scope of DBA.

The key ingredients of the formalism are the following: a DBA program manipulates a finite set of global variables ranging over bitvectors (registers) and an array of bitvectors of size 8 (memory); all bitvector sizes are statically known; a single machine instruction is decoded into a *block* of DBA instructions - including intermediate computations and temporary variables.

Extended DBA model. While DBA have shown to be useful in the analysis of safety-critical systems [1], they lack abstraction and specification mechanisms in order to handle binary-level analysis over large non-critical codes [1]. We propose the following improvements:

– more abstract operations (`malloc`, `nondet`) together with basic specification mechanisms (`assume`, `assert`), see Figure 1;
– a more abstract low-level region-based semantics [2], representing memory as a dynamic collection of disjoint arrays (`constant`, `stack`, `malloc(id)`) while being able to simulate precisely many low-level programs, see Section 4;
– access permissions for `read`, `write` and `execute` operations; permissions are defined on *region zones*, i.e. region partitions defined by (user-given) predicates;
– tags on instructions and variables for embedding useful information available at decoding, such as `<tmp>` or `<flag>` for variables and `<call>` or `<ret>` for jumps.

Instructions
- `lhs := rhs, goto addr`
- `goto addr` $<$ `call, ret, none` $>$
- `goto expr` $<$ `call, ret, none` $>$
- `ite(cond)? goto addr : goto addr′`
- `lhs := malloc(size), goto addr`
- `free(expr), goto addr`
- `lhs := nondet(size), goto addr`
- `assert(cond), goto addr`
- `assume(cond), goto addr`
- `stop` $<$ `ok, ko, none` $>$

Expressions
- $\texttt{e\{i .. j\}}$, $\texttt{ext}_{u,s}\texttt{(e,n)}$, $\texttt{e} :: \texttt{e}$
- $\texttt{@(expr, } \overrightarrow{k}\texttt{)}$, $\texttt{@(expr, } \overleftarrow{k}\texttt{)}$
- $\texttt{e} \ \{+, -, \times, /_{u,s}, \%_{u,s}\} \ \texttt{e}$
- $\texttt{e} \ \{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\} \ \texttt{e}$
- $\texttt{e} \ \{\wedge, \vee, \oplus, <<, >>_{u,s}\} \ \texttt{e}$, `!e`

Fig. 1: Extended DBA instructions

3 Platform overview

BinSec is designed around three basic services, depicted in Figure 2: (1) a front-end translating executable codes into DBA programs (loading, decoding, syntactic disassembly, support of DBA stubs) and simplifying them; (2) a simulator for extended DBA, supporting three different memory models (flat, standard regions [8], low-level regions [2]); and finally (3) a generic static analysis engine (in progress) allowing safe CFG recovery.

[1] This drawback is common to other formal IRs such as REIL [9], RREIL [13] and BAP [7].

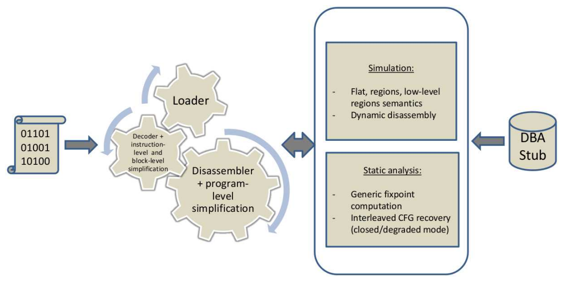
01101
01001
10100
Loader
Decoder + instruction-level and block-level simplification
Disassembler + program-level simplification
Simulation:
- Flat, regions, low-level regions semantics
- Dynamic disassembly
Static analysis:
- Generic fixpoint computation
- Interleaved CFG recovery (closed/degraded mode)
DBA Stub

Fig. 2: BINSEC platform

3.1 Front-end

Loading and decoding. The main service here is a decoding function taking a (virtual) address and returning a block of DBA instructions simulating the semantics of the corresponding machine code instruction. The platform currently supports the `ELF` format (a `PE` loader is in progress) and a decoder for `x86-32` is provided. The following features are supported: all prefixes but `wait` and `lock`, all basic instructions (380 instr.) and all `mmx` instructions not involving floating-point registers (100 instr.).

Disassembly. The goal of disassembly is to give the semantics of the whole executable file. This is a very hard problem because of dynamic jumps [6, 10, 11]. We provide implementations of the most common solutions: (1) recursive disassembly, with the possibility to specify some jump targets; (2) linear sweep disassembly (typically used by `objdump`) with instruction-wise or byte-wise granularity, the later allowing to disassemble overlapping instructions; (3) a combination of recursive and linear sweep disassembly, mimicking the approach of `IDA pro`; and finally (4) a combination of recursive disassembly with dynamic execution, where jump targets are discovered through simulation.

Formal stubs. A formal stub is a block of DBA instructions that will be inserted at some address of the retrieved program, either in place of the result of the decoder (`@replace`) or in combination with it (`@insert`). This feature is useful either when the corresponding code is not available (analysis of object files rather than executable files), or for abstracting parts of the code (typically, library functions). A stub for `libc/malloc` function is described in Figure 3.

```
@replace :
0xb7fff414 {
        tmp<32> := nondet(32);                       // abstracting a failure condition, typically out of memory
        if (tmp = 0<32>) goto l1 else goto l2;
   l1 : eax<32> := 0<32>;  goto l3;                                              // failure, result is NULL
   l2 : eax<32> := malloc(@[esp + 4<32>,<-,4]);                 // DBA malloc, with size read on stack
        assume ((eax modu 4<32>) = 0<32>);                                      // alignment constraint
   l3 : esp<32> := esp + 4<32>;                                                      // stack cleanup
        goto @[esp - 4<32>,<-,4];                 // jump to return address (call-site) retrieved from the stack
}
```

Fig. 3. A stub for `libc/malloc`

3.2 Simplifications

Simplifications discard unused DBA instructions, typically those instructions modeling flag updates. The goal is to help later analyzes, either automatic or human-based. We essentially try to simplify temporary variables and flag variables, identified through DBA tags. We rely on rewriting rules (instruction-wise), constant propagation and elimination of temporary variables (block-wise), and liveness analysis for flag elimination (inter-block). The method removes up to 75% of flag operations (cf. Section 4).

3.3 Memory model and simulation

Memory models. We provide a partitioned memory model in the vein of CompCert [8], with values of the form (r, val) where r is a region symbol - the base, and val is a bitvector - the offset (Cst being a special region symbol acting as 0). This modeling is very adapted for managing dynamically allocated memory and allows robust formal analyzes thanks to implicit partitioning of memory. However, most operations are illegal with pure regions [8], e.g. $(r_1, v_1) - (r_2, v_2)$ is undefined when $r_1 \neq r_2$ and $r_2 \neq Cst$. Unfortunately, undefined patterns are found in common `libc` programs, such as `memmove` or `memcopy`, and, even worst, they can also be introduced at compile-time. For instance, an instruction `x = if (!x) then 1 else 0;` can be compiled as follows (assuming `x` is stored in `eax`):

```
neg eax              // eax := -eax.  CF := 0 if source operand (eax) is 0; otherwise CF := 1
sbb eax, eax         // eax := eax - (eax + CF) = -CF
inc eax              // eax := eax + 1 = -CF + 1
```

The compiler performs here an optimization called *branchless conditional* in order to optimize instruction pipelining. In a region-based model, the result of the first `neg` instruction is undefined when the input is a pointer value, i.e. $r \neq Cst$. Low-level region-based models [2] have been introduced recently to address this issue by allowing some reasoning over region symbols.

Simulation. We provide simulation and random testing modes supporting all features of extended DBA. Three different memory models can be selected: (a) flat model (memory as a single array), (b) standard region-based model and (c) low-level region-based model. Interestingly, all models are implemented in a unified way, pure regions and flat model being viewed as restrictions of low-level regions.

3.4 Static analysis interface

We provide a generic fixpoint computation for abstract domains given as lattices, allowing one to quickly prototype binary-level analyzers. The current implementation offers the following advantages: (1) tight interleaving of syntactic disassembly with value analysis [6, 11], allowing sound resolution of indirect jumps; (2) the possibility to restrict a priori the set of possible jump targets (closed mode) by providing a finite set of acceptable targets; (3) a degraded mode, in the vein of [10], where the analyzer switches to an unsound analysis whenever a jump or a memory operation cannot be resolved precisely enough. The interface is currently limited to non-relational abstract domains. We plan to extend it quickly to relational domains and to provide implementations of the most common domains.

4 Experiments

We evaluate our implementation on two main criteria: the impact of low-level regions and the effectiveness of our simplifications. Simplifications are performed over standard Unix programs, while experiments on low-level regions are carried upon a collection of small- to medium-size procedures (up to 5,000 machine instructions) from `libc` and the VeriSec benchmark[2]. All experiments are performed on an Intel Core i5 3.20Ghz.

Benefits of simplifications. Results are presented in Table 1 and summarized in Table 2. Simplifications allow a global reduction of instructions of 24%, and (most important) flag assignments are reduced by about 73%, which is interesting because these operations are complex to handle in analyzers. Simplified DBA programs are in average 2.5x larger than native codes (3.3x larger without simplifications) [3]. This is pretty close to the minimal ratio between DBA and machine code, since an inter-block `goto` is added to each DBA block.

program	native	DBA		simplified DBA		
	loc	loc	† ko	loc	time	red
bash	166K	558K	5	402K	10.65m	27.95%
cat	7303	23K	0	18K	16.62s	20.55%
echo	3345	10K	0	8181	6.39s	22.38%
less	23K	80K	5	56K	89.31s	29.03%
ls	18K	63K	6	45K	83.42s	27.38%
mkdir	7329	24K	5	18K	23.65s	27.08%
netstat	16K	50K	3	41K	68.48s	17.43%
ps	11125	36K	0	28K	47.90s	21.38%
pwd	3581	11K	0	8942	9.77s	21.47%
rm	9186	30K	16	23K	31.13s	22.52%
sed	9993	32K	0	24K	37.50s	24.24%
tar	64K	212K	7	159K	5.2m	25.26%
touch	7944	26K	0	19K	30.02s	25.75%
uname	3271	10K	0	8131	8.89s	21.68%

† ko: # unsupported instructions

Table 1. Evaluating DBA optimization.

DBA vs asm (no simpl)		3.3x
reduction	dba instr	24.00%
	tmp assign	21.89%
	flag assign	73.17%
DBA vs asm (simpl)		2.5x

DBA vs asm: ratio between # DBA instructions and # machine instructions

Table 2. Average reductions

Benefits of low-level regions. We compare both memory models on their ability to provide defined concrete semantics on the benchmark programs. These programs contain some patterns that illustrate illegal operations in standard region-based model. Results are summarized in Table 3, where we also provide time information w.r.t. the flat memory model. The standard region-based model succeeds in only 1/20 example, while low-level regions succeed in 20/20 examples. It seems that low-level regions are absolutely necessary in order to give a (useful) non-flat semantics to binary programs.

[2] Available at `https://se.cs.toronto.edu/index.php/Verisec_Suite`.

[3] Simon *et al.* report a 7x size increase for GDSL/RREIL, and a 3.5x size increase after simplifications [12].

program	standard regions	low-level regions	flat
aligned_calloc	x	✓ 4.73s	0.0003s
llpointer_arithmetic	x	✓ 3.51s	0.01s
malloc	x	✓ 0.62s	0.008s
memcpy	x	✓ 0.001s	0.003
memmove	x	✓ 0.49s	0.01s
mmap	x	✓ 0.03s	0.02s
neg_sbb_inc	x	✓ 2.81s	2.82s
pointer_arithmetic	x	✓ 0.02s	0.02s
pointer_logical	x	✓ 0.12s	0.001
pointer_or_int	x	✓ 0.07s	0.0006s
success	0/10	10/10	10/10

program	standard regions	low-level regions	flat
test_or_pointer	1.08s	✓ 1.09s	1.09s
loops	x	✓ 1.006s	1.07s
full	x	✓ 5.76s	5.73s
istrstr	x	✓ 5.54s	5.77s
istrstr_loops	x	✓ 5.40s	5.61s
istrstr2_loops	x	✓ 5.27s	5.64s
parse_config	x	✓ 3.83s	4.12s
guard_random_index	x	✓ 0.14s	0.13
guard_strstr	x	✓ 5.53s	5.53s
guard_strchr	x	✓ 2.98s	3.02s
success	1/10	10/10	10/10

Table 3. Simulation with three different memory models

5 Future Work

We plan to extend very quickly our framework with more decoders (`PowerPC`, `ARM`) and loaders (`PE`). We also plan to extend the static analysis interface and add basic facilities for symbolic execution, taking low-level memory regions into account.

References

1. Bardin, S., Baufreton, P., Cornuet, N., Herrmann, P., Labbé, S.: Binary-level Testing of Embedded Programs. In: QSIC 2013. IEEE, Los Alamitos (2013)
2. Blazy S., Besson F., Wilke P.: A Precise and Abstract Memory Model for C using Symbolic Values. In: APLAS 2014. Springer, Heidelberg (2014)
3. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: ICST 2008. IEEE, Los Alamitos (2013)
4. Bardin, S., Herrmann, P.: OSMOSE: Automatic Structural Testing of Executables. Softw. Test., Verif. Reliab. 21(1): 29-54(2011)
5. Bardin S. , Herrmann P., Leroux J., Ly O., Tabary R., Vincent A.: The BINCOA Framwork for Binary Code Analysis. In: CAV 2011. Springer, Heidelberg (2011)
6. Bardin S., Herrman P., Védrine F.: Refinement-based CFG Reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
7. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.: BAP: A Binary Analysis Platform. In: CAV 2011. Springer, Heidelberg (2011)
8. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model. In: Program Logics for Certified Compilers. Cambridge University Press (2014)
9. Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest 2009.
10. Kinder, J., Kravchenko, D.: Alternating Control Flow Reconstruction. In: VMCAI 2012. Springer, Heidelberg (2012)
11. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: CAV 2008. Springer, Heidelberg (2008)
12. Simon, A., Kranz, J.: The GDSL toolkit: Generating Frontends for the Analysis of Machine Code. In: PPREW 2014. ACM, New York (2014)
13. Sepp, A., Mihaila, B., Simon A.: Precise Static Analysis of Binaries by Extracting Relational Information. In: WCRE 2011. IEEE, Los Alamitos (2011)