The BINCOA Framework for Binary Code Analysis [⋆]

Sébastien Bardin[1], Philippe Herrmann[1], Jérôme Leroux[2], Olivier Ly[2], Renaud Tabary[2], and Aymeric Vincent[2]

(1) CEA, LIST,
Gif-sur-Yvette CEDEX,
F-91191, France
first.last@cea.fr

(2) LaBRI
351 cours de la Libération
33405 Talence Cedex
first.last@labri.fr

Abstract. This paper presents the BINCOA framework, whose goal is to ease the development of binary code analysers by providing an open formal model for low-level programs (typically: executable files), an XML format for easy exchange of models and some basic tool support. The BINCOA framework already comes with three different analysers, including simulation, test generation and Control-Flow Graph reconstruction.

1 Introduction

Automatic analysis of programs from their executable files is a recent and promising field of research, opening the way to new applications of software verification (mobile code, off the shelf components, legacy code, malware) and more accurate and reliable analyses (taking into account the compilation step). In the last years, a few teams have been involved in this emerging research field and a few techniques and tools have been developed [1, 3, 5–10, 13, 14], mostly based on static analysis and symbolic execution.

The problem. Besides specific theoretical challenges, binary code analysis suffers from two major practical issues. First, implementing a binary code analyser requires lots of programming efforts. There exist many different instruction set architectures (ISA), and each ISA counts several dozens of instructions. Hence adding support for a new ISA is a time-consuming, tedious and error-prone activity. Moreover, it follows that each analyser supports only very few ISAs, making different technologies and tools difficult to compare. Second, each analyser comes with its own formal model of binary code. Since the exact semantics is seldom available, modelling hypotheses are often unclear and may differ from one tool to the other, making results and models difficult to reuse.

The BINCOA framework. We describe in this paper the BINary COde Analysis (BINCOA) framework, whose aim is to ease the development of binary code analysers.

1- The framework is constructed around Dynamic Bitvector Automata (DBA), a generic and concise formal model for low-level programs. The main design ideas behind DBA are the following: (a) a small set of instructions; (b) a concise and natural modelling for common architectures; (c) self-contained models which do not require a

⋆ Work partially funded by ANR grant ANR-08-SEGI-006.

separate description of the memory model or of the architecture; and (d) a sufficiently low-level formalism, so that DBA can serve as a reference semantics of the executable file to analyse. Most features of low-level programs are taken into account by this formalism, including dynamic jumps, modification of the call stack, instruction overlapping and endianness. The two main limitations are the following: the formalism cannot capture self-modifying code and it is untimed.

2- We intend to gather an ecosystem of binary code analysers around DBA, all tools being able to share their front-ends and exchange their results. To this end, we have defined an XML DTD to communicate DBA and we provide open-source code for basic DBA manipulation, including XML input/output and DBA simplifications.

3- DBA are already used by three different analysers: Osmose [3, 4] for test generation, TraceAnalyzer for safe Control-Flow Graph (CFG) reconstruction (based on [6]) and Insight, a platform providing front-end, simulation and some value analysis mechanism. Altogether, these three tools prove that DBA can encode a few different architectures and ISAs (including PowerPC and x86).

Why using DBA and the BINCOA framework? The BINCOA framework eases the development of binary code analysers: (semantics) all analysers built upon DBA can be fairly compared and their results can be safely reused from one tool to the other; (engineering) the BINCOA framework provides open-source basic facilities for DBA manipulations. In the future we also plan to provide an open-source platform allowing to share front-ends and ISA support.

Moreover, we think that DBA are a good trade-off between conciseness and ease of use: there are only about two dozen of operators (to be compared with any ISA) and modelling common ISAs with DBA is straightforward. DBA have already been used to encode four different ISA, including x86 and PowerPC.

Outline. The remaining part of the paper is structured as follows: the DBA model is described in Section 2, its practical usage is discussed in Section 3, tool support for DBA and different analysers based on DBA are presented in Section 4, finally Section 5 describes related work and Section 6 provides a conclusion and some future work.

2 DBA in a nutshell

The syntax and semantics of DBA is sketched hereafter. A detailed description can be found in the technical report [2].

DBA are automata extended with a finite set of variables ranging over fixed-width bit-vectors and a finite set of (disjoint) fixed-size arrays of bytes (bit-vectors of size 8). Some of the nodes of the automaton are labelled with addresses ranging over $\mathbb{N}$. Transitions are decorated with basic instructions: assignments (**Assign** lhs := rhs), no-operation (**Skip**), guards (**Guard** cond), jumps to a non-statically known address (**Jump** expr), and an instruction to handle absent code such as API calls (**External** φ). The first three kind of instructions are standard. The **External** instruction, followed by a first-order formula over bit-vectors and arrays, allows to introduce a non-deterministic computation step defined in a pre/post-condition style. The **Jump** instruction is described hereafter. The operational semantics is given by a transition system in a standard manner. Expressions and conditions are built upon a small set of standard fixed-width bit-vector operators, including (signed / unsigned) arithmetic operators, reified (signed / unsigned) arithmetic relational operators, logical bitwise operators, size extensions, shifts, concatenation and restriction. Contrary to real processor instructions, these operators are side-effect free. Every expression evaluates to a bit-vector of statically known size (this is not a restriction considering current ISAs).

Original features. DBA provide a few original mechanisms dedicated to low-level languages. (1) Dynamic jumps (**Jump** transitions) are *dangling*, i.e. they do not have a predefined target node. When the transition is fired, the jump expression is evaluated and turned into an integer a, and the control-flow goes to the node labelled by a. Dynamic jumps are necessary for modelling indirect branching. (2) Bit manipulations can be expressed easily thanks to the restriction operator, available both in lhs and rhs operands. (3) Multiple-byte read and write operations (and incidentally endianness) can be expressed easily thanks to a dedicated array-access operator of the form: array$[expr; k^{\#}]$, where $k \in \mathbb{N}$ and $\# \in \{\leftarrow, \rightarrow\}$, denoting the k consecutive indices starting at index $expr$ and accessed in big-endian ($\rightarrow$) or little-endian ($\leftarrow$). (4) Memory zone properties define specific behaviours for segments of arrays; currently available properties are *write-is-ignored*, *write-aborts*, *read-aborts* and *volatile* (with their intuitive meanings).

A few remarks. (1) The set of operators in DBAs is not minimal, however we think that it is a nice trade-off between conciseness and ease of use. (2) Realistic programs have typically only a few dynamic jumps, hence "realistic" DBAs will behave mostly as standard extended automata. This motivates the choice of an automata-based formalism for DBA. (3) DBAs do not have native support for procedure calls and returns: they are encoded as jumps, as it is the only correct semantic for call/return instructions found in ISAs. However, since this extra information may be useful if treated with care, the XML format for DBA (see Section 4) allows to annotate jumps with call/return tags.

3 Modelling low level programs with DBA

Basics. Most architectures and ISAs can be modelled accurately using the following rules. Each register in the processor is modelled by a variable in the automaton, additional variables ("local" variables) may be introduced to encode ISA instructions needing intermediate results, e.g. for side-effects. An ISA instruction at address a is translated in at least one node labelled by a and one transition. Additional ("local") nodes and transitions may be needed for intermediate computations. The additional nodes are not labelled. A single array is usually sufficient for memory. Additional arrays may allow for example to distinguish an I/O bus from a memory bus. Memory zone properties can be used for ROM (*write-is-ignored*), code section of the program (*write-aborts*, allowing to detect self-modifying code) or memory controlled by an external device (*volatile*). Instructions with side effects (e.g. flag updating) are split into several DBA instructions by adding local nodes and transitions.

Figure 1 presents a few examples of ISA instruction modelling through DBA. We suppose that each ISA instruction is encoded on four bytes. The ISA instruction is on the left column, and the corresponding DBA is on the right. ISA instructions are supposed to be located at address `0x5003` in the executable file. For the second example (an addition instruction), we suppose that the instruction updates a carry flag `Fc` (the carry-flag is set to 0 iff the *unsigned* addition is correct).

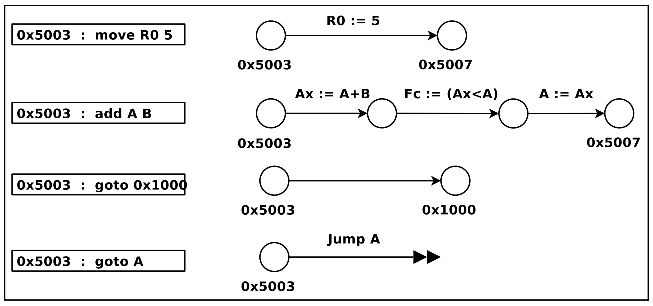

Fig. 1. DBA encoding of a few typical instructions

Open programs and interruptions. DBA provides various ways to model programs interacting with an external environment, either hardware (sensor/actuator) or software (OS). A sequential interaction may be simulated either by a stub or by a logical specification (**External**). A concurrent interaction may be simulated in the general case by a product of DBA, and in simple cases by declaring a volatile memory zone.

Limitations. The two strongest limitations of DBA are that they are untimed and that they cannot encompass self-modifying code. There are also a few “weaker” limitations, i.e. mechanisms with no native support, which can still be modelled in DBA but at a possibly high cost. These limitations are mainly dynamic memory (de-)allocation, run-time modification of endianness and asynchronous interruptions. Finally, DBA do not provide any operator for floating-point arithmetic. It is straightforward to add them to the model, but taking them into account in the analysers is much more demanding.

4 The BINCOA Framework

Tool support for DBA manipulation. We provide open-source OCaml code for basic DBA manipulation[1]. The module contains a datatype for DBA, import / export functions from / to the XML format defined in the technical report [2], as well as type checking (based on bit-vector sizes) and simplification functions for a few typical inefficient patterns observed in automatic DBA generation (typically, removing useless flag computations). These simplifications are inspired by standard code optimisation techniques (peephole, dead code elimination, etc.), and are adapted to be sound on *partial* DBAs, in case where the DBA is recovered incrementally from an executable file. We observed a reduction from 10% to 55% of the number of DBA instructions with these simplifications. The XML parser is based on xml-light [2] and the library counts about 3 kloc. The code is under GPL license.

[1] `https://bincoa.labri.fr/`
[2] `http://tech.motion-twin.com/xmllight`

Insight: decoding, simulation and analysis platform.[3] Insight is a platform developed mostly in C++ and offers the ability to load executable files supported by the GNU BFD library and disassemble them with a homebrew disassembler. Its internal representation is very close to DBA and import/export of DBA in XML is possible. It offers a general setting for concrete and symbolic execution of the model, as well as a generic annotation facility which allows to prove assertions using weakest precondition computation. The platform currently has three satellite tools allowing to disassemble a program, to execute it concretely or symbolically (intervals, sets of values, probability distributions), and to apply control flow graph reconstruction to polymorphic virus analysis.

Osmose: test data generation. Osmose [3, 4] is a test data generation tool for binary code, based on dynamic symbolic execution and bit-vector constraint solving. The tool also offers test suite replay via a simulation engine, test suite completion, (unsafe) test suite coverage estimation, (under-approximated) CFG recovery and a graphical user interface. Front-ends are available for PowerPC, Intel 8051 and Motorola 6800. The tool can export / import DBA given in XML. The program contains 75 kloc of OCaml. A few industrial case-studies have been successfully carried out.

TraceAnalyzer: safe and precise CFG recovery. TraceAnalyzer performs safe and precise CFG reconstruction from an executable file. The core technology is a refinement-based static analysis [6]. The program is about 29 kloc of C++. A front-end for PowerPC is available, as well as import / export facilities from / to DBA.

A concrete example of cooperation between tools. TraceAnalyzer and Osmose are able to communicate in two ways. First, Osmose can receive from TraceAnalyzer an upper approximation of every set of jump targets and take advantage of it to provide a safe coverage measure, which is crucial for example in critical system certification. Second, TraceAnalyzer can receive from Osmose a set of observed jump targets, and take advantage of it to efficiently bootstrap its refinement-based static analysis.

5 Related work

Many binary code analysers have been developed recently, for example to name a few: CodeSurfer/x86 [1], Sage [8], Bitscope [5], Osmose [3], Jakstab [9] and McVeto [14]. However most of them are based on a "private" formal model, with no available specification. We are aware of two other generic low-level models suitable for executable analysis, but none of them is open. DBA can be seen as the successor of the Generic Assembly Language (GAL) of Osmose [4], which is similar to DBA in both goals and shape. However DBA are more concise, easier to manipulate and more expressive than GAL. Actually, GAL shows a few shortcomings that have been addressed in DBA: no loops in intermediate nodes, no native support for endianness, unduly complex operators with multiple return-values. Osmose is being redesigned to work on DBAs instead of GAL. TSL [12], developed by Lim and Reps to re-implement CodeSurfer/x86 [1], is based on semantic reinterpretation: each instruction of the ISA is given a concrete semantic written in a ML-like language (with only a limited set of basic operators); adding a new analysis is mainly done by overloading every basic operator. This is rather similar

[3] `http://insight.labri.fr/`

to the idea behind BINCOA, the ML description serving as the reference model. TSL and DBA should have more or less the same modelling power, however comparison is difficult since TSL is not publicly available.

LLVM [11] is a generic low-level language designed for compilation rather than verification. Hence LLVM abstraction level is in between binary code and C: it provides many low-level operations, as well as higher level features based on the knowledge of the initial source code (types, native array manipulation).

6 Conclusion and perspectives

This paper presents the BINCOA framework for binary code analysis. BINCOA aims at easing the development of binary code analysers by providing an open formal model (DBA) for low-level programs, an XML format to allow easy exchange of both models and benchmarks and some basic tool support. Future work comprises providing more open-source support (visualisation tools, x86 and ARM front-ends) as well as extending DBA with native support for memory allocation and facilities for self-modifying code.

References

1. Balakrishnan, G., Gruian, R., Reps, T., W., Teitelbaum, T.: CodeSurfer/x86-A Platform for Analyzing x86 Executables. In: CC 2005. Springer, Heidelberg (2005)
2. Bardin, S., Fleury, E., Herrmann, P., Leroux, J., Ly, O., Sighireanu, M., Tabary, R., Touili, T., Vincent, A.: Description of the BINCOA Model. Deliverable J1.1 part 2 of ANR Project BINCOA (2009). Available at `https://bincoa.labri.fr/`
3. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
4. Bardin, S., Herrmann, P.: OSMOSE: Automatic Structural Testing of Executables. International Journal of Software Testing, Verification and Reliability (STVR), 21(1), 2011
5. Brumley, D., Hartwig, C., Kang, M., G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: BitScope: Automatically Dissecting Malicious Binaries. Carnegie Mellon Uni. technical report CS-07-133. CMU (2007)
6. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG Reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
7. Balakrishnan, G., Reps, T., W.: Analyzing Memory Accesses in x86 Executables. In: CC 2004. Springer, Heidelberg (2004)
8. Godefroid, P., Levin, M., Y., Molnar, D., A.: Automated Whitebox Fuzz Testing. In: NDSS 2008. The Internet Society (2008)
9. Kinder, J., Veith, H.: Jakstab: A Static Analysis Platform for Binaries. In: CAV 2008. Springer, Heidelberg (2008)
10. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: VMCAI 2009. Springer, Heidelberg (2009)
11. Lattner, C.: The LLVM Compiler Infrastructure Project. `http://llvm.org/`.
12. Lim, J., Reps, T., W.: A System for Generating Static Analyzers for Machine Instructions. In:. CC 2008. Springer, Heidelberg (2008)
13. Reps, T., Lim, J., Thakur, A., Balakrishnan, G., Lal, A. There's plenty of room at the bottom: Analyzing and verifying machine code. In: CAV 2010. Springer, Heidelberg (2010)
14. Thakur, A., Lim, J., Lal, A., Burton, A.,Driscoll, E.,Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: CAV 2010. Springer, Heidelberg (2010)