BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis[⋆]

Robin David,
Sébastien Bardin
CEA LIST,
Software Safety and Security Laboratoy
Saclay, France
first.last@cea.fr

Thanh Dinh Ta,
Josselin Feist,
Laurent Mounier,
Marie-Laure Potet
Vérimag, Grenoble, France
first.last@imag.fr

Jean-Yves Marion
Université de Lorraine,
CNRS and Inria, LORIA, France
jean-yves.marion@loria.fr

***Abstract*—When it comes to software analysis, several approaches exist from heuristic techniques to formal methods, which are helpful at solving different kinds of problems. Unfortunately very few initiative seek to aggregate this techniques in the same platform. BINSEC intend to fulfill this lack of binary analysis platform by allowing to perform modular analysis. This work focusses on BINSEC/SE, the new dynamic symbolic execution engine (DSE) implemented in BINSEC. We will highlight the novelties of the engine, especially in terms of interactions between concrete and symbolic execution or optimization of formula generation. Finally, two reverse engineering applications are shown in order to emphasize the tool effectiveness.**

I. Introduction

While security can be checked and enforced at different level, we focus here on binary-level security analyses, such as malware comprehension or vulnerability analysis. Reverse engineering of binary codes is a key component of these activities, yet it is notoriously difficult. Especially, low-level assembly constructs can deceive disassemblers, while dynamic execution explores only a few possible behaviors. Our long-term goal is to adapt formal methods, which have been very successful in source-level safety analysis, to binary-level security analysis. We present in this paper a binary-level dynamic symbolic execution engine, named BINSEC/SE, geared toward security analysis and especially reverse-engineering.

The main contribution is a highly configurable generic DSE engine toolkit, with a strong interaction between the tracer and the symbolic execution core, as well as heavy optimizations on the path predicate. Two reverse engineering applications are shown to emphasize the tool effectiveness.

II. Background

Dynamic Symbolic Execution [5], [9] is a formal technique for exploring program paths in a systematic way. For each path, the technique computes a *path predicate*, i.e. a set of constraints on the program input that leads to follow that path at runtime. This predicate is then fed to an automatic solver: a solution to the predicate is a new test input exploring the targeted path. Systematic exploration is achieved through iterating on all (user-bounded) paths of the program.

BINSEC [8] is a recent platform for formal analysis of binary codes. The platform currently proposes a front-end from `x86(32bits)` to a generic intermediate representation called DBA [8] (including decoding, disassembling, simplifications), a simulator and a static analysis engine (typically for dynamic jump resolution). The platform is written in OCaml *(∼30,000 loc)*, and will be released under an open source license soon[1]. Our DSE engine BINSEC/SE is built upon BINSEC. The current tool paper describes only BINSEC/SE, a consequent add-on to BINSEC reusing only the `x86`-to-DBA translation.

III. User view

From a user point of view, the advantage of using a DSE engine is twofold. First, nowadays closed source binaries and malwares are so commonly widespread that having tools for working at binary level is mandatory. Second, recovering information from a binary can be a tedious task, especially in heavily obfuscated binaries. Therefore applying automatic techniques for recovering information is crucial with regard to the usual amount of code to review.

During a reverse-engineering task or an information retrieval task, the analyst can be interested in knowing information about the binaries, e.g. the register value at a given location, the possible values at a given memory address, whether a given branch can be covered or not or the possible targets of a dynamic jump. These problems can be addressed by DSE.

Various dynamic analyses are already implemented in BINSEC/SE and new ones can conveniently be written using a callback mechanism. An analysis takes a JSON configuration file as input and a trace file. Both the configuration and trace file formats are open and specified.

IV. BINSEC/SE architecture

We describe hereafter the new module for DSE, named BINSEC/SE. Its implementation is made of three components, presented in Figure 1.

- a pintool named PINSEC, written in C++ *(∼2,000 loc)*, based on the Pin DBI framework [10] for dynamic tracing;

[⋆] Work partially funded by ANR, under grant ANR-12-INSE-0002

[1] http://binsec.gforge.inria.fr

- the core DSE engine written in OCaml*(∼8,000 loc)*, whose tasks are to generate path predicates, to send them to a solver and to get back new input data;
- the path selector, written in OCaml*(∼1,200 loc)*, in charge of choosing which path to explore next.

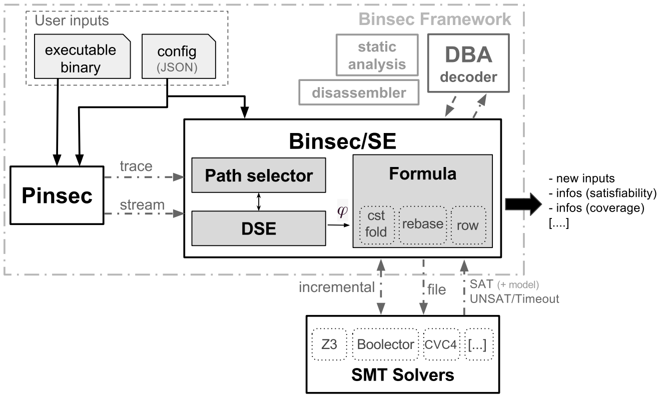


Fig. 1. BINSEC/SE architecture

A. The PINSEC tracer

The tracer, called PINSEC, is a pintool [10] specialized in analyzing `x86` architectures. Its main purpose is to generate the execution trace used as input by BINSEC. It is designed to offer a generic and modular tracing for both Linux and Windows binaries, and to export results into a generic format, here `protobuf`[2] which is usable with major programming languages. The tracing is parameterized via a JSON file whose format is also defined in `protobuf`. The main parameters are:

- `start` and `stop`: addresses indicating where to begin and where to end the tracing,
- `call_skips`: address of calls for which the callee should not be traced,
- `fun_skips`: function addresses that should not be traced.

Other command line parameters allow to limit the trace size, the tracing time or the instrumentation scope.

Advanced usage. What is differentiating PINSEC from other pintools are the following functionalities:

- the automatic retrieval of function parameters and return values for (some) libraries functions;
- the possible retrieval of the concrete value of any register or memory location, even if not part of the operands / results of the currently instrumented instruction;
- the injection of symbolic or concrete values in any register or memory location, at any step of the execution;
- a remote command and control system allowing the DSE core and PINSEC to exchange messages in an interactive, debugger-like manner, in order to dynamically tune the instrumentation.

A full-fledged example using some of these parameters is presented in Section VI.

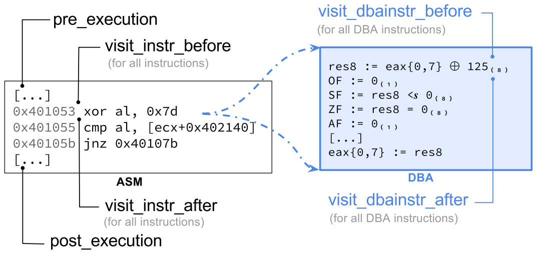
B. DSE engine

Our DSE engine (Figure 1) follows a classical workflow. From an input trace, x86 instructions are translated into DBA instructions [8] (such a translation can be seen in Figure 2) on which the path predicate is computed [12], and then used to generate a formula (theory of arrays and bitvectors) which is then exported to the SMTLIB2 format[3], the standard input of modern SMT solvers. A SMT solver is finally used to check the satisfiability of the formula and if so, to get back a solution (new input). We currently rely on Z3, CVC4 and Boolector. The connexion between BINSEC/SE and PINSEC easily allows to re-inject the new input to get a new trace.

Implementation insight. Internally an analysis is represented as an OCaml class, which each analysis should inherit from. This class provides appropriate callbacks and data structures for implementing any trace-based analysis. These callbacks allow to apply specific actions at a specific step and/or location along the execution in a highly configurable manner. They help developing new analyses without a deep understanding of the whole inner-working of the DSE. Main callbacks are:

- `pre_execution`, `post_execution` triggered respectively once at the begining and the end of the analysis;
- `visit_instr_before`, `visit_instr_after` triggered before (resp. after) every asm instruction of the trace;
- `visit_dbainstr_before`, `visit_dbainstr_after` triggered before (resp. after) every DBA instructions of an x86 instruction;
- `input_message_received` triggered when a message is received from PINSEC.

A summary of all callback location calls is given in Figure 2.



```
[...]
0x401053 xor al, 0x7d
0x401055 cmp al, [ecx+0x402140]
0x40105b jnz 0x40107b
[...]
```

```
res8 := eax{0,7} ⊕ 125(8)
OF := 0(1)
SF := res8 <s 0(8)
ZF := res8 = 0(8)
AF := 0(1)
[...]
eax{0,7} := res8
```

Fig. 2. BINSEC/SE callbacks

C. Path selection

Beside a fine-grained control of a single trace execution, automatic multiple paths exploration is also a desired feature of a DSE tool. Ideally, the path exploration engine should allow either to fulfill some standard coverage requirements, or to focus on specific parts of the code through dedicated (user-defined) search heuristics. The design of the exploration module of BINSEC/SE is inspired from the one of OSMOSE [2]. It relies on a simple API allowing to easily implement various exploration strategies. In particular this API offers a function `select(S)` returning the “best” trace from a set of traces `S`, w.r.t. a user-defined `score` function. Scores

[2] `https://developers.google.com/protocol-buffers/`

[3] `http://smtlib.cs.uiowa.edu/`

are built from several (predefined or dynamically computed) trace characteristics, such as length, last instruction call-depth, distance to a given target, etc. This approach allows to precisely define a wide-range of exploration strategies. Several strategies are already implemented such as DFS, BFS, random path, MinCall-DFS and MinCall-BFS [1].

V. Formula generation and solving

The bottom-line of DSE is path predicate generation. While we follow the standard approach [12], i.e. maintening a symbolic memory state along the computation together with the encountered path constraints (i.e. branching conditions), our method is original in several aspects:

- we can take into account constraints on the initial memory state (cf. Section V-B);
- we provide callback mechanisms for fine-tuning of concretization and symbolization (cf. Section V-C);
- the path predicate is highly optimized, and we take advantage of incremental solving (cf. Section V-A).

A. Optimizations

The core engine implements several optimizations of the path predicate, ranging from standard simplifications such as constant folding, local rewriting (e.g. $a \oplus a$ becomes 0) or pruning useless parts of the formula, to less common optimizations, such as Read-over-Write (RoW) simplifications over array load and store [4]. Finally, great care is taken in order to make these optimizations compatible with the incremental solving mode of modern SMT solvers, which is particularly well-suited to DSE, since path predicates are naturally built incrementally.

Table I shows the results of optimization on a set of malware samples (cf. Section VI-B), while Table II shows a more detailed view on a single trace taken from the Artelad malware. The timeout is set to 20 seconds on a Intel Core i7 2.7 GHz (16GB RAM) using the Z3 solver.

TABLE I
Optimization benchmarks

Malware	#inst	#br (#queries)	#Solved	no-Incr			Incr		
				no-opt	pruning	full-opt	no-opt	pruning	full-opt
Artelad	10K	1295	1295	42m36	31m27	18m18	12m34	11m57	9m6
Benny	10K	3150	3510	**14m35**	10m34	7m54	3.64	2.58	**2.94**
Bogus	10K	191	191	**41m51**	7.71	7.34	1.45	2.98	**3.41**
Cornad	404	32	21	4m10	4m10	4m5	4m43	4m40	4m37
Eva	1069	115	115	**3.44**	2.07	1.98	0.16	0.15	**0.16**
Htrip	2525	1410	1410	**15m6**	12m23	11m34	18.36	18.52	**11.17**
Total	303998	6193	6181	118m25	58m48	42m03	17m46	17m3	14m3
Avg/query				1.15	0.57	0.40	0.16	0.16	0.14

time in seconds, or minutes/seconds **Avg/query**: average time per query
full-opt: all optimization turned on – **no-opt**: all optimizations turned off
Incr: incremental solving – **no-Incr**: non-incremental solving

B. Initial memory state

Logical arrays are unconstrained by default. Hence, modelling the initial memory state as a logical array implies that the solver is free to give any value to any memory location of the initial state, possibly leading to meaningless results since the initial mapping of addresses in a real program is more complex. BINSEC/SE allows to specify constraints on the initial memory state (initial logical array).

TABLE II
Effects of optimizations

	no-opt	pruning	full-opt
#VarsDef	27375	8952	3026
#Constraints	1585	1581	379
#Memory operations	1589	1585	1136
#Operators	85805	6330	2278

C. Concretization/Symbolization

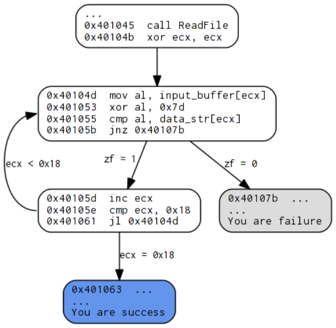
For scalability issues, the path predicate is often approximated, either by forcing some logical values to their runtime values (observed by the tracer) – *concretization*, or by injecting fresh logical input – *symbolization*. Both operations reduce the complexity of the formula, at the price of completness and/or correctness. So choosing appropriately when and what to concretize/symbolize is a major practical issue. While most tools offer only hardcoded policies, BINSEC provides a complete set of callbacks to specify such choices. Such a mechanism allows, for instance, symbolic reasoning on memory pointers, where existing tools usually enforce memory addresses to be concrete.

VI. Experimentations

A. Reverse engineering

In reverse-engineering, so-called `crackme` challenges simulate real world situations where binary programs are designed to make the analysis difficult. The goal is usually to find a string key (the flag) allowing to validate the challenge.

Flare-On is a reverse engineering challenge organized since 2014 by FireEye Security[4]. We analyse the first challenge of 2015 since it is straightforward enough to be discussed in detail. This crackme is a 32 bits Windows program which asks for a password and prints “You are success” or “You are failure” depending on the keyboard input. Each byte of the input (`input_buffer`) is xored with `0x7d`, the result is then checked against a key stored in the data section (`data_str`), cf. Figure 3.



```
...
0x401045  call ReadFile
0x40104b  xor ecx, ecx
```

```
0x40104d  mov al, input_buffer[ecx]
0x401053  xor al, 0x7d
0x401055  cmp al, data_str[ecx]
0x40105b  jnz 0x40107b
```

```
0x40105d  inc ecx
0x40105e  cmp ecx, 0x18
0x401061  jl 0x40104d
```

```
0x40107b  ...
...
You are failure
```

```
0x401063  ...
...
You are success
```

Fig. 3. Flare-on #1 key loop computation

Solving the crackme by symbolic execution aims at checking that the predicate `ZF=1` is true at location `0x40105b`; and this, for every character of the key. If the generated formula is

[4] `http://www.flare-on.com/`

satisfiable, then the right character can simply be retrieved in the formula solution at `input_buffer[ecx]`. One last remaining problem is the initial state. If none is specified, the solver is allowed to give any valuation to the content of `data_str`, while it is not user-controllable. Hence, an initial state constraining the value of the `data_str` bytes is required in order to get meaningful results for `input_buffer` (cf. SectionV-B). Finally, iterating over `input_buffer` can be done in two ways:

1) From a trace taking the fail branch, we should solve `ZF=1` at `0x40105b` to get the right char and inject it back as input via the configuration file for generating a new trace that will do a second loop iteration, repeating until the whole key is found;
2) Configure a breakpoint at `0x40105b` (the address of `jnz`), compute the right character value, send a command to patch the `ZF` flag and resume to force the tracer to take the right branch and looping again. This method is fully automatic and allow to take advantage of incremental solving (cf. Section V-A).

We obtain the right key solving the challenge, `bunny_sl0pe@flare-on.com`[5]. Note that we have solved several other Flare-on challenges with the same principle.

B. *Malware exploration*

We are interested here in demonstrating the ability of BINSEC/SE to automatically discover new code areas of a program, which is especially crucial in malware analysis. We consider 11 malware programs taken from the VX Heaven database[6] and used in recent works on deobfuscation [11]. Table III shows our results. We report the number of new explored paths (i.e. each time DSE manages to negate a branch of the original trace), and of new behaviors (i.e. each time DSE covers a branch not yet covered by the initial trace or by a generated one). Here, we are able to discover 43 new code areas in the malwares under consideration.

TABLE III
MALWARE EXPLORATION

File	#inst	#br	#uniq	#path (new)	#behavior (new)	Avg	Total
Virus.Win32.Artelad.2173	10K	1295	3	652	2	0.59	759.72
Virus.Win32.Belial.a	10K	1551	3	602	2	0.10	152.94
Virus.Win32.Benny.3219.a	10K	3153	121	4	4	0.0	3.10
Virus.Win32.Bogus.4096	10K	191	1	0	0	0.02	4.45
Virus.Win32.Cabanas.a	10K	2269	479	1	1	0.00	8.72
Virus.Win32.Cornad	404	32	17	12	12	0.63	20.05
Virus.Win32.Eva.a	1069	115	49	4	4	0.01	0.87
Virus.Win32.Htrip.a	2525	1105	9	598	6	0.10	137.23
Virus.Win32.Pulkfer.a	10K	1526	9	766	6	2.25	3434
Virus.Win32.Seppuku.1606	1696	147	61	4	4	0.01	1.85
Virus.Win32.Wit.a	10K	3334	2	3334	2	3.00	9985.96
Total	75694	14718	754	5977	43	6.71	14508.89

br: number of branches – **uniq**: unique conditional jumps – **path**: number of new paths – **behavior**: new path covering a new code area – **Avg**: Average solving time per paths

VII. Related work

A few other initiatives aim at applying DSE on binary programs for security purposes, most notably BAP [3], S2E [7], FuzzBall [6] or Triton [13]. All these approaches also rely on an intermediate representations similar to DBA. Regarding dynamic tracing, these tools are built either on Qemu or on Pin. In addition to a generic DSE engine, the novelties provided by BINSEC/SE are highly configurable path exploration strategies and concretization/symbolization policies, a strong interaction between the tracer and the DSE core and an optimized path predicate computation.

VIII. Conclusion

The dynamic symbolic execution implemented in the platform yields the interesting property of being modular w.r.t the analysis configuration and to provide a strong interaction between the DSE (BINSEC) and the tracer (PINSEC). Some functionalities like the formula optimizations makes it original from the existing state-of-the art tools. This platform comes really handy for reverse-engineering and information recovery tasks like it is the case for source-less programs and malwares.

References

[1] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbé. Binary-level testing of embedded programs. In: QSIC 2013. IEEE, 2013.

[2] S. Bardin and P. Herrmann. OSMOSE: Automatic Structural Testing of Executables. *Software Testing, Verification and Reliability*, 21(1), 2011.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A Binary Analysis Platform. In: CAV 2011. Springer, 2011.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In: CCS 2006. ACM, 2006.

[5] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *CACM*, 56, 2013.

[6] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, and D. Song. Transformation-aware Exploit Generation using a HI-CFG. Technical report, University of California, Berkeley, 2013.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In: ASPLOS XVI. ACM, 2011.

[8] A. Djoudi and S. Bardin. BINSEC: Binary Code Analysis with Low-Level Regions. In: TACAS 2015. Springer, 2015.

[9] P. Godefroid, M. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 2012.

[10] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: PLDI 2005. ACM, 2005.

[11] M. H. Nguyen, M. Ogawa, and T. Q. Thanh. Obfuscation code localization based on CFG generation of malware. In: FPS 2015. Springer, 2015.

[12] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution. In: SP 2010. IEEE, 2010.

[13] http://triton.quarkslab.com/.

[5] A complete demo is available at `https://youtu.be/0xUc2jbpjQo`

[6] http://vxheaven.org/