BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis
Robin David, Sébastien Bardin, Thanh Ta, Josselin Feist, Laurent Mounier,
Marie-Laure Potet, Jean-Yves Marion

To cite this version:
Robin David, Sébastien Bardin, Thanh Ta, Josselin Feist, Laurent Mounier, et al.. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. 3rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Mar 2016, Osaka, Japan. hal-01721502

HAL Id: hal-01721502
http://hal.univ-grenoble-alpes.fr/hal-01721502
Submitted on 2 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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.

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.

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

1http://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](image)

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.

![Fig. 2. BINSEC/SE callbacks](image)

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 beginning 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.

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. maintaining 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

<table>
<thead>
<tr>
<th>Malware</th>
<th>#Var</th>
<th>#Sce</th>
<th>MinCall-DFS</th>
<th>MinCall-BFS</th>
<th>#VarsDef</th>
<th>#Constraints</th>
<th>#Memory operations</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artelad</td>
<td>10K</td>
<td>1290</td>
<td>42m58</td>
<td>7m31</td>
<td>124s58</td>
<td>11s87</td>
<td>596</td>
<td>1410</td>
</tr>
<tr>
<td>Bogos</td>
<td>10K</td>
<td>3150</td>
<td>114m37</td>
<td>7m63</td>
<td>51m45</td>
<td>27m54</td>
<td>7.71</td>
<td>145</td>
</tr>
<tr>
<td>Corna</td>
<td>#Not</td>
<td>#Not</td>
<td>44m34</td>
<td>25m30</td>
<td>60m45</td>
<td>#Not</td>
<td>#Not</td>
<td>#Not</td>
</tr>
<tr>
<td>Visa</td>
<td>1500</td>
<td>115</td>
<td>4m44</td>
<td>5m09</td>
<td>60m45</td>
<td>60m40</td>
<td>68s31</td>
<td>2525</td>
</tr>
<tr>
<td>Total</td>
<td>#Not</td>
<td>#Not</td>
<td>42m58</td>
<td>25m30</td>
<td>58m46</td>
<td>#Not</td>
<td>#Not</td>
<td>#Not</td>
</tr>
</tbody>
</table>

**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).

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 completeness 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. 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.

### Table II: Effects of Optimizations

<table>
<thead>
<tr>
<th>#VarsDef</th>
<th>no-opt</th>
<th>pruning</th>
<th>full-opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>27375</td>
<td>8952</td>
<td>3026</td>
<td></td>
</tr>
<tr>
<td>1585</td>
<td>1581</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>1589</td>
<td>1585</td>
<td>1136</td>
<td></td>
</tr>
<tr>
<td>85805</td>
<td>6330</td>
<td>2278</td>
<td></td>
</tr>
</tbody>
</table>

**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

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. Section V-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.com5. 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 database8 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>
<thead>
<tr>
<th>File</th>
<th>Brct</th>
<th>Br</th>
<th>Uniq Jumps</th>
<th>Path (new)</th>
<th>Behavior (new)</th>
<th>Avg</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virus.Win32.Antiad.2173</td>
<td>10K</td>
<td>1295</td>
<td>3</td>
<td>652</td>
<td>2</td>
<td>0.59</td>
<td>759.72</td>
</tr>
<tr>
<td>Virus.Win32.Bblea.x</td>
<td>10K</td>
<td>1551</td>
<td>3</td>
<td>602</td>
<td>2</td>
<td>0.35</td>
<td>152.94</td>
</tr>
<tr>
<td>Virus.Win32.Benny.3219.x</td>
<td>10K</td>
<td>3153</td>
<td>121</td>
<td>4</td>
<td>4</td>
<td>0.6</td>
<td>3.10</td>
</tr>
<tr>
<td>Virus.Win32.Bogus.4996</td>
<td>10K</td>
<td>191</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.02</td>
<td>4.95</td>
</tr>
<tr>
<td>Virus.Win32.Cabanas.x</td>
<td>10K</td>
<td>2609</td>
<td>409</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>8.92</td>
</tr>
<tr>
<td>Virus.Win32.Cornad</td>
<td>804</td>
<td>12</td>
<td>17</td>
<td>12</td>
<td>12</td>
<td>0.63</td>
<td>2010</td>
</tr>
<tr>
<td>Virus.Win32.Eve.s</td>
<td>1097</td>
<td>115</td>
<td>49</td>
<td>4</td>
<td>4</td>
<td>0.08</td>
<td>0.87</td>
</tr>
<tr>
<td>Virus.Win32.Hippe.x</td>
<td>2322</td>
<td>1105</td>
<td>9</td>
<td>660</td>
<td>6</td>
<td>0.38</td>
<td>123.23</td>
</tr>
<tr>
<td>Virus.Win32.Plattel.x</td>
<td>10K</td>
<td>1520</td>
<td>9</td>
<td>506</td>
<td>6</td>
<td>2.23</td>
<td>313.60</td>
</tr>
<tr>
<td>Virus.Win32.Seppeka.1600</td>
<td>1496</td>
<td>147</td>
<td>61</td>
<td>4</td>
<td>4</td>
<td>0.05</td>
<td>1.95</td>
</tr>
<tr>
<td>Virus.Win32.Wd.x</td>
<td>10K</td>
<td>3334</td>
<td>2</td>
<td>3334</td>
<td>2</td>
<td>0.08</td>
<td>3983.96</td>
</tr>
<tr>
<td>Total</td>
<td>17504</td>
<td>1819</td>
<td>784</td>
<td>507</td>
<td>63</td>
<td>6.31</td>
<td>14089.19</td>
</tr>
</tbody>
</table>

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 path

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