Original software publication

**SENinja: A symbolic execution plugin for Binary Ninja**

Luca Borzacchiello *, Emilio Coppa, Camil Demetrescu

*Sapienza University of Rome, Italy*

**A R T I C L E I N F O**

**Article history:**
Received 5 October 2020
Received in revised form 16 December 2021
Accepted 26 September 2022

**Keywords:**
Reverse engineering
Symbolic execution
Cybersecurity

**A B S T R A C T**

Symbolic execution is a program analysis technique that aims to automatically identify interesting inputs for an application, using them to generate program executions covering different parts of the code. It is widely used in the context of vulnerability discovery and reverse engineering. In this paper we present SENinja, a symbolic execution plugin for the BinaryNinja disassembler. The tool allows the user to perform symbolic execution analyses directly within the user interface of the disassembler, and can be used to support a variety of reverse engineering tasks.

© 2022 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

**1. Motivation and significance**

Software reverse engineering is the process of reconstructing the operation, the design, and the architecture of a piece of software, starting from an end product, e.g., a compiled binary program. The process is typically hard since it involves analyzing thousands of lines of code, written in low-level languages (e.g., assembly), without documentation and often obfuscated to be harder to analyze. Despite the difficulties, reverse engineering is crucial in several circumstances: for example, in malware analysis and security assessment of proprietary software.

While reverse engineering is mostly a manual task, researchers and developers have built tools and techniques that can help to speed up the process. Disassemblers are essential tools for analyzing compiled binary programs. The job of a disassembler is to translate a compiled binary into human-readable assembly code, arranging it in a Control-Flow Graph (CFG) that highlights the structure of the code. There are several available disassemblers [1–4], and among them, BinaryNinja [5] is one of the most used by the cybersecurity community. In addition to the normal tasks of a disassembler, it implements other types of analyses and exposes them in a complete and well-documented set of APIs. For example, BinaryNinja performs code lifting, which is the translation of assembly code of a given architecture to a higher-level intermediate language (IL). Examples of such languages are LLVM IR [6] and VEX [7]. Lifting simplifies program analysis as it: (a) reduces the number of different (often redundant) instructions that need to be handled by an analysis and (b) favors portability since any architecture supported by the lifter will be also handled by the analysis. BinaryNinja lifts instructions of the most common architectures (e.g., x86, x86_64, ARM, MIPS) to LLIL (Low Level IL): Fig. 1(b) shows on the right the LLIL generated by BinaryNinja when lifting the x86_64 code shown on left.

Symbolic execution is a widely popular technique in the context of bug detection and reverse engineering [8–14] that can automatically generate inputs for a program. The goal is achieved by constructing expressions over symbolic inputs and using a satisfiability modulo theory (SMT) solver (e.g., Z3 [15], FuzzySAT [16]) to reason over them. As an example, consider Fig. 1(a). On the
left, we have a function authenticate\(^1\), while on the right we have the symbolic tree that represents the result of the symbolic exploration on this function. A symbolic execution engine evaluates the code of a function as an interpreter, initializing input variables as symbols (in the example, variable \(a\) is initialized with symbol \(α\) which can assume initially any value in the interval \([0, 2^{32} - 1]\)), and building symbolic expressions instead of performing computations on concrete values. A state is the abstract object that holds the memory and the constraints accumulated in an execution path. When the execution hits a branch, if the condition involves symbolic values, the execution forks, i.e., the symbolic engine splits the current state into two states. The two states model the outcomes of the two branch directions (in the example, line 3 generates two states to model when \(α \oplus 170 \neq 187\) is true and false, respectively). The execution can continue on the two states separately. At any time during the exploration, the constraints collected in a state can be used to generate, with the help of an SMT solver, an input that would have driven a concrete execution along the same path of the state. In Fig. 1(a), the two final states in the execution tree can be reproduced using input values equal to \(α = 0\) and \(α = 17\) respectively. Notice that it is very unlikely that a brute-force approach would generate an input that covers line 6, since the search space has \(2^{32}\) values.

Symbolic execution has proven to be a fundamental ingredient for finding bugs and vulnerabilities. For instance, it was used during the development of Windows 7, finding almost one-third of the bugs revealed with fuzzing techniques [17]. Moreover, it has been also a pivotal component for most systems playing in the Cyber Grand Challenge [18], a two-year competition run by DARPA seeking to create automated tools for finding, exploiting, and patching software vulnerabilities.

2. Software description

In this article, we present SENinja, a tool that implements a symbolic execution engine as a plugin of BinaryNinja. SENinja evaluates the Low Level IL (LLIL) generated by BinaryNinja and is integrated into the BinaryNinja user interface (UI), allowing users to perform symbolic execution without switching to other tools. Fig. 4 gives a visual overview of the plugin.

2.1. Software architecture

Fig. 2 shows the architecture of SENinja. The main software component of the tool is the Executor. It is a high-level interface that is in charge of holding the generated states and of executing instructions symbolically on the current active state. It interacts with BinaryNinja to obtain crucial information about a binary, such as the LLIL representation and the memory layout. The commands exposed by SENinja, that are accessible through the UI of BinaryNinja, are constructed using this high-level interface.

In the next sections, we describe in more detail the inner components of the Executor, explaining some of the design choices that we made.

2.1.1. State

A state represents a snapshot of the execution for a path. Looking at the right-hand side of Fig. 1(a), every node in the

---

1 The function is written in C for simplicity; SENinja targets binary code.
A well-known problem in symbolic execution is state explosion. While SENinja cannot solve this problem in general, it can at least minimize the overhead of keeping track of different but similar states generated during the exploration. To this aim, we have designed every component of the state to have a Copy-on-Write (CoW) behavior in order to reduce resource consumption when forking a state.

Another common problem in symbolic execution is the handling of symbolic memory accesses, i.e., reasoning on the effects of a memory operation when the memory address depends on the value of the program inputs. SENinja supports different memory models for handling memory accesses:

**Fully symbolic.** Symbolic memory accesses are handled by considering every memory cell that can be accessed. While this is the slowest mode, it is also the most accurate.

**Fully concrete.** This model concretizes the expression of the address to a single concrete value. This is the fastest mode, but also the less accurate.

**Partially symbolic.** This model falls in the middle of the previous approaches. It uses a fully symbolic approach, but only if the number of possible values that the symbolic address can assume is sufficiently small. If the address is concretized, the access is correctly handled to a newly allocated page and any other symbolic address referring it as a base address is handled accurately within the allocated page. This is the default memory model in SENinja.

To evaluate the impact of the symbolic memory models and the CoW strategy, we consider a benchmark involving a symbolic computation of a CRC32 checksum. The left chart of Fig. 3 shows the running time of different symbolic executors when computing the checksum on an increasing number of symbolic bytes (from 1 to 1024 bytes). The benchmark is characterized by several symbolic accesses, whose result is crucial to compute the input that when processed should generate an expected CRC value. We consider:

(a) *Klee* [10], a source-based symbolic executor,
(b) *angr* [8], a binary symbolic framework, enabling the support for symbolic accesses,
(c) SENinja (fully concrete), which uses the fully concrete memory model, and
(d) SENinja (partially symbolic), which uses the partially symbolic memory model. We do not consider the fully symbolic memory model in this benchmark since the memory accesses are restricted within a few memory pages, thus generating the same behavior as the partially symbolic memory model.

SENinja (fully concrete) is very efficient but very inaccurate: it fails (cross markers in the chart) to derive the input for most checksum sizes. *angr* scales only for small checksum sizes (up to 16 bytes), as then it takes more than 1 hour (which was the timeout during our experiment). *Klee* is very efficient, however, it exploits knowledge derived from the source code (in particular, the size of an array accessed by the benchmark). SENinja (partially symbolic) can correctly reason on the checksum computation up to 512 bytes, being faster than *Klee* for several checksum sizes. Recently proposed array optimizations could be integrated into SENinja to further improve its scalability.

The middle and right charts of Fig. 3 show the resource consumption of SENinja (partially symbolic) with and without the CoW strategy. During these experiments, we have disabled the solver to focus on the resource consumption due to state exploration, which is what is impacted by the CoW strategy. The benefits resulting from the CoW strategy can be clearly seen in terms of running time and memory consumption.
2.1.3. Symbolic expressions

SENinja represents symbolic expressions using the theory of bitvectors [28], which models the semantics of fixed-size bitvectors arithmetic. In particular, SENINJA uses a custom Abstract Syntax Tree class to wrap bitvector objects from the Z3 SMT solver. It does not use directly the AST of Z3 for mainly two reasons: (a) concrete computations can be performed more efficiently and (b) SENINJA can be easily ported to other SMT solvers by updating the wrapper class. Additionally, SENINJA enriches the AST representing an expression with a range interval, that provides an over-approximation on the possible values that an expression can assume in a state. For instance, SENINJA computes the interval range $[256, 512]$ given the expression $256 + \alpha$, which represents a 32-bit addition of the constant 256 to a zero-extended 8-bit input value $\alpha$. Interval analysis is extremely valuable in the presence of symbolic memory accesses as it may allow SENINJA to evaluate which memory pages could be modified during the execution without querying an SMT solver. The current implementation does not yet support strided intervals and in case of wrap-around returns the range $[0, 2^n - 1]$, where $n$ is the number of bits in the expression.

2.1.4. OS and function models

To handle system calls and invocations to functions from dynamic libraries, SENINJA devises models [8] that describe the effects of external code on the current state. Currently, SENINJA provides models for the most common C library functions (e.g., `memcpy`, `memset`), and the most used Linux system calls. The models are written in Python, and new models can be added with a few lines of code [29]. However, to reduce the need of writing OS models from scratch, SENINJA offers preliminary support for a compatibility layer that allows it to reuse models available for the well-known symbolic executor ANGR [8].

Finally, SENINJA supports custom hooks [30]. They allow modeling a small part of the functionalities of an external piece of code, which is sufficient in several reverse engineering tasks and can be used to overcome the lack of some models.

2.2. Tool functionalities

Fig. 4 shows an overview of the interface of SENINJA. We now review the main functionalities, highlighting how they can be accessed directly through the UI of BinaryNinja.

**Symbolic state construction and initialization.** The symbolic execution can start at any point in the program. SENINJA initializes a state using the memory content obtained from BinaryNinja. It also exploits the Value Set Analysis [31] performed by BinaryNinja to detect, e.g., constant registers. By default, unknown data is marked as symbolic, however a user can choose other policies (e.g., zero-initialization).

**Debugger-like step functions.** In SENINJA only a single state can be active at any time. Symbolic execution can be performed on the current active state using commands that are inspired by debuggers. The commands are: `single step`, `continue until address` and `continue until branch`. Hence, through the UI, the user can

---

Fig. 4. The BinaryNinja interface with the SENINJA plugin. The active state is at the address in green (1). Deferred states are marked in red (2), showing a comment to indicate the number of states at the same address. The memory and registers of the active state can be viewed using widgets (3) and (4). Symbolic buffers can be viewed and created using (5). Commands are accessible through the right-click menu (6). The CLI can be accessed using the Python console (7).
change the current active state and start a new exploration using one of the previous commands.

Since the symbolic exploration may take a long time to, e.g., reach a specific address, the user can bound the exploration time by setting a timeout (through the panel settings), or stop the exploration at any time using a dedicated command from the right-click menu.

After an exploration, SENinja can highlight in the CFG which instructionshave been executed by a state during the exploration.

**State merging.** If two or more states are executing the same instruction, the user can decide to merge them [33]. While state merging can reduce memory consumption, the solver may struggle in reasoning on formulas derived from a merged state, since they can be more complex.

The merging algorithm is inspired by the strategy implemented in the source-based symbolic executor Klee [10]. Before merging two states, SENinja checks their successors: if they are different, i.e., the two states would take different directions, then the merging operation is aborted.

**Automatic searchers.** In addition to executing a single state, SENinja devides automatic searchers that can be used to search through the paths of the program in order to find an input that reaches a certain program point. The user, through the right-click menu, can set an address as the target of the search and can mark a set of addresses to be avoided during the search. Then the user can start the searching process using a DFS or BFS algorithm.

**Memory, register and buffer view.** The memory and the registers of the current active state can be viewed using the SENinja widgets (see (3) and (4) in Fig. 4). The widgets can be used to view and modify concrete data, view symbolic expressions, evaluate expressions using the solver, or inject new symbols. When evaluating an expression, the user can generate up to $k$ solutions, where $k$ is a user-defined value. Symbolic buffers can be created and constrained using a dedicated widget (see (5) in Fig. 4).

**Command line interface.** Complex operations can be performed using the command-line interface. BinaryNinja has an embedded Python console, which can be used to invoke the command-line API of SENinja. For example, the user can set specific constraints over an input, or can define a custom hook for a library function. A detailed description of the command-line API can be found in the project wiki.

3. Illustrative example: analyzing virtual machine obfuscation

In this section, we present one case study in which we use SENinja for reverse engineering of obfuscated code.
Obfuscation is the act of producing code that is difficult to understand by a human. Developers obfuscate code in order to make the reverse engineering process more difficult, e.g., to protect a license checker or a proprietary algorithm. Obfuscation is also widespread among malware writers.

Virtual machine obfuscation is one of the most used and effective obfuscation techniques [34]: it translates the code to obfuscate into a custom byte code and then replaces the original code in the binary with the bytecode and a custom virtual machine that at runtime is able to reproduce the behavior of the original code when interpreting with custom opcode handlers the generated bytecode. As an example of obfuscated code, we consider the 11th challenge [32] from the reverse competition Flare-On 6 [35]. The program is a 64-bit PE that uses virtual machine obfuscation to protect a function that checks several conditions on user-provided inputs. Hence, this function could be seen as a license key checker and we use SENinja to automatically find inputs that are accepted by this checker.

3.1. Preliminary analysis

We begin by manually analyzing the binary using BinaryNinja. The main function can be identified at address 0x140001220 (see Fig. 5). This function considers two input strings (obtained as command-line arguments), where the second string has a size of 32 bytes. It then calls vm_loop: this function is the virtual machine dispatcher loop, i.e., the routine that fetches the bytecode and calls the proper handlers to perform the obfuscated computation. After running the obfuscated code, main calls function final_checks, which checks that the first string is FLARE2019 and validates the output of the obfuscated computation, executing the code at 0x14000169d in case of success or the code at 0x14000178a in case of failure. Since the first input is known after this preliminary analysis, the main goal is to find the value of the second input without spending hours manually reversing the obfuscated computation.

3.2. Finding a valid input

After obtaining a general idea of the structure of the binary, we can use SENinja to automatically identify a value for the second input able to satisfy the check. We first create an initial state at beginning of main (right-click, Start symbolic execution). Then we use the buffers widget to create a new symbolic buffer of 32 bytes (step 1 in Fig. 6). We then set up the command-line arguments using the Setup argv command from the SENinja toolbar (step 2), setting the string FLARE2019 as the first argument and the buffer that we just created as the second argument.

After defining the symbolic inputs and creating an initial state, we define address 0x14000169d as the target point (right-click, Set target) in the code that we want to reach during the symbolic exploration and address 0x14000178a as an avoid point (right-click, Set avoid) in the code that is not interesting for our exploration. Finally, we can start the execution exploiting the DFS searcher (right-click, run DFS).

After a few seconds, SENinja is able to generate a state reaching the target point. Using the buffers widget (steps 3 and 4 in Fig. 6), we can obtain the concrete input that passes the check: cHcyrAHSmxEkpyqoCBCyGhuyFyCmy86Ee.

4. Comparison with other tools

A few previous works [36,37] have explored solutions for integrating symbolic execution into graphical reverse engineering tools.

For instance, Ponce [36] integrates the dynamic symbolic execution engine Triton [38] into the commercial disassembler and debugger IDA Pro. A crucial design difference with SENinja is that Ponce cannot analyze code statically, which is a common requirement in presence of binaries for non-standard architectures, or non-executable memory dumps.

Another interesting solution is IDAngr [37], which combines the symbolic framework angr [8] with IDA Pro. Unfortunately, this plugin is not actively maintained anymore and the integration with the UI of IDA Pro is quite limited.

AngryGhidra [39] and modality [40] are two recent projects that expose the functionalities of angr in Ghidra [4] and Radare2 [2], respectively. AngryGhidra is designed to obtain some exploration parameters (e.g., the starting target) from the user through the UI but then it starts angr using a fixed and predefined script, leaving very limited opportunity for interactions. Modality instead embraces the spirit of Radare2 and exposes
several new actions in its command-line interface. Several steps from Section 3 cannot be performed when using the current release of these two plugins, forcing the user to manually interact with ANGR or to face severe path explosion.

Finally, SymNav [41] devises a visual representation of the symbolic tree. Unfortunately, this viewer is a standalone component that cannot be currently integrated into debuggers, such as IDA PRO or BinaryNinja.

5. Impact and conclusions

SENinja is a symbolic execution plugin for BinaryNinja, a commercial disassembler widely used by the cybersecurity community. SENinja extends the functionalities of the disassembler, giving the user access to symbolic execution analysis directly within BinaryNinja, possibly simplifying reverse engineering activities. Furthermore, it is designed to be extensible, allowing users to implement new features by typically adding a few lines of Python code.

After the public release of SENinja on GitHub, the community of BinaryNinja has shown a positive interest in it: SENinja has been recently officially included in the community plugin repository [42] of BinaryNinja. Moreover, a well-known security expert has tried SENinja, positively mentioning it in a blog post [43]. We hope that, in the next few years, SENinja can become one of the reference tools for reverse engineers.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2022.101219.

References