SymNav: Visually Assisting Symbolic Execution

Marco Angelini,* Graziano Blasilli,* Luca Borzacchiello,†
Emilio Coppa,* Daniele Cono D'Elia,* Camil Demetrescu,*
Simone Lenti,* Simone Nicchi,* Giuseppe Santucci*
Sapienza University of Rome

Abstract

Modern software systems require the support of automatic program analyses to answer questions about their correctness, reliability, and safety. In recent years, symbolic execution techniques have played a pivotal role in this field, backing research in different domains such as software testing and software security. Like other powerful machine analyses, symbolic execution is often affected by efficiency and scalability issues that can be mitigated when a domain expert interacts with its working, steering the computation to achieve the desired goals faster. In this paper we explore how visual analytics techniques can help the user to grasp properties of the ongoing analysis and use such insights to refine the symbolic exploration process. To this end, we discuss two real-world usage scenarios from the malware analysis and the vulnerability detection domains, showing how our prototype system can help users make a wiser use of symbolic exploration techniques in the analysis of binary code.

Index Terms: Human-centered computing—Visualization—Visualization application domains—Visual Analytics; Security and privacy—Software and application security—Software reverse engineering

1 Introduction

Analyzing software programs is hard. To seek answers regarding aspects such as determining software defects, understanding the semantics of a code portion, verifying when a certain program property holds, or checking whether a bug may represent a vulnerability are complex tasks and the use of machine analyses to reason over software is not only desirable, but nowadays unavoidable.

In the realm of program analyses well in vogue in recent years, symbolic execution techniques back hundreds of research works and industrial applications in many security and software testing scenarios [9]. In the context of cybersecurity research, symbolic execution has gained popularity as a powerful means to analyze programs in order to, e.g., discover vulnerabilities, bypass authentication or protection schemes, or understand their internals.

A prominent success story is the DARPA Cyber Grand Challenge, held in 2016, where autonomous systems competed with each other in discovering, patching, and exploiting vulnerabilities in never-seen-before software. Symbolic execution played a crucial role in the competition, supporting prominent participating tools in the software analysis endeavor. By exploring multiple execution paths in a program, the technique can apply, in principle, a number of decidable analyses looking for, e.g., certain classes of pointer bugs that may be exploited by an attacker.

In spite of the flexibility of the technique, which stems from maintaining a detailed view of the machine state across multi-path executions and using satisfiability modulo theory (SMT) solvers to reason over symbolic inputs, symbolic execution is fraught with many pitfalls that limit its practical applicability. Most prominently, it can be severely affected by the combinatorial explosion of the execution space deriving, e.g., from branches in the program. The state space explosion can slow down the analysis process to the point where no answer is found for any realistic resource budget [9]. While these issues already appear in software testing applications, the situation gets even worse in several security scenarios.

For starters, the code to be analyzed is typically available in binary form only, making it harder for security analysts to interpret the results of symbolic execution, but also for analysis systems as they cannot access semantic information that gets typically lost during compilation (e.g., types, array bounds). The analyst may also be seeking deeper answers: let us consider vulnerability detection scenarios, where analyses focus not only on detecting bugs, but also on checking if they are exploitable, allowing for instance an attacker to hijack the control flow of the program. As if it were not enough, the analysis of untrusted code such as malware samples generally incurs an additional complexity layer, having to face adversarial code often fraught with obfuscated operations and anti-analysis techniques that can mislead symbolic execution or SMT solver reasoning.

Being able to effectively monitor the actions performed by a symbolic execution engine and steer it away from uninteresting code regions is of paramount importance in security applications where purely automated analyses are likely to fail. Unfortunately, to date the interaction with existing engines is mainly based on lengthy textual information that provides feedback on the different active execution branches and their computation states, imposing a heavy burden on the analyst.

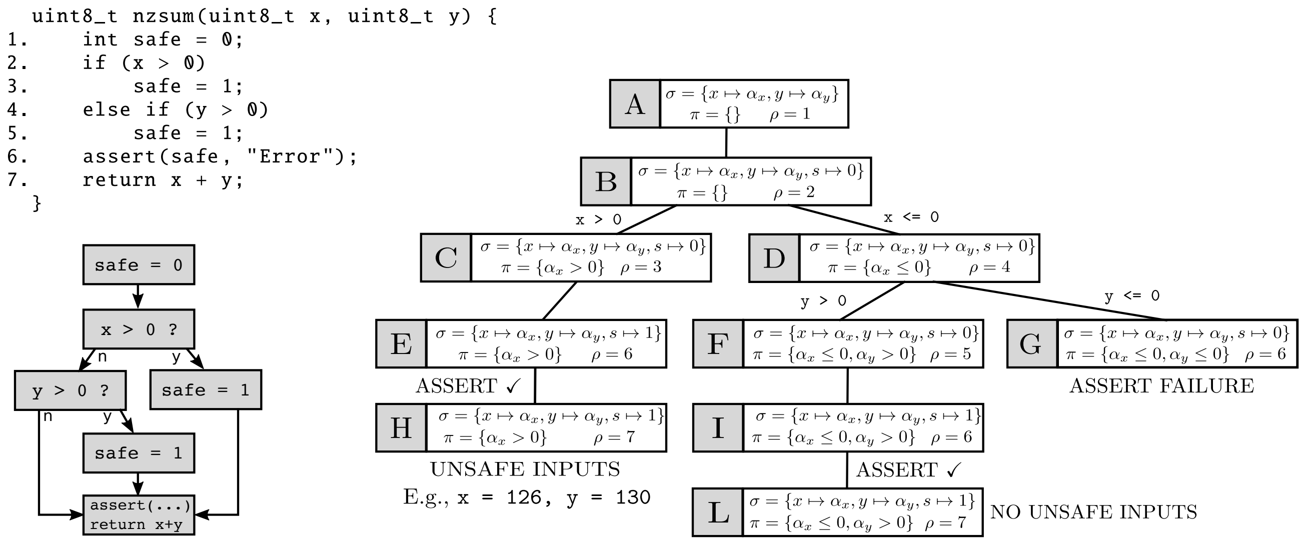
Contributions. In this paper, we explore how a Visual Analytics approach can assist security analysts in the complex task of monitoring and steering analyses based on symbolic execution, allowing them to deal with complex explorations that would be too cumbersome to address using conventional textual information. In particular, the contributions of the paper are the following:

- it introduces a novel visual analytics environment targeted at shedding light on different aspects of the exploration performed by a symbolic execution engine when analyzing a program during an experiment;
- it explores several analytical and visual solutions to assist the user in understanding which program parts have been explored by the symbolic engine, pinpointing branches that led to a large number of states and the inputs involved in decision points;
- it lets analysts leverage the visual feedback to steer the computation by prioritizing the execution towards promising paths;
- it presents two usage scenarios on using the solution for the analysis of complex malware and software vulnerabilities.

2 Application Domain

Technique. Symbolic execution is a program analysis technique for program property verification and has seen hundreds of applications in software testing and software security research. The idea underpinning this technique is to have a program take symbolic rather than concrete input values, supporting the exploration of mul-

*e-mail: {angelini, blasilli, borzacchiello, coppa, delia, demetres, lenti, nicchi, santucci}@diag.uniroma1.it
†Corresponding author (e-mail: borzacchiello@diag.uniroma1.it).



```
  uint8_t nzsum(uint8_t x, uint8_t y) {
1.    int safe = 0;
2.    if (x > 0)
3.        safe = 1;
4.    else if (y > 0)
5.        safe = 1;
6.    assert(safe, "Error");
7.    return x + y;
  }
```

safe = 0
x > 0 ?
n
y
y > 0 ?
safe = 1
n
y
safe = 1
assert(...)
return x+y
A $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \{\}$ $\rho = 1$
B $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 0\}$ $\pi = \{\}$ $\rho = 2$
x > 0
x <= 0
C $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 0\}$ $\pi = \{\alpha_x > 0\}$ $\rho = 3$
D $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 0\}$ $\pi = \{\alpha_x \leq 0\}$ $\rho = 4$
y > 0
y <= 0
E $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 1\}$ $\pi = \{\alpha_x > 0\}$ $\rho = 6$
F $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 0\}$ $\pi = \{\alpha_x \leq 0, \alpha_y > 0\}$ $\rho = 5$
G $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 0\}$ $\pi = \{\alpha_x \leq 0, \alpha_y \leq 0\}$ $\rho = 6$
ASSERT ✓
ASSERT FAILURE
H $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 1\}$ $\pi = \{\alpha_x > 0\}$ $\rho = 7$
I $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 1\}$ $\pi = \{\alpha_x \leq 0, \alpha_y > 0\}$ $\rho = 6$
UNSAFE INPUTS
E.g., x = 126, y = 130
ASSERT ✓
L $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y, s \mapsto 1\}$ $\pi = \{\alpha_x \leq 0, \alpha_y > 0\}$ $\rho = 7$
NO UNSAFE INPUTS

Figure 1: Introductory example: code and control flow graph (left) and symbolic execution tree (right).

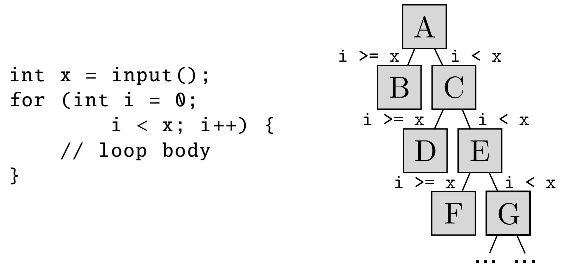
tiple possible executions at once. Each path followed by a symbolic execution engine represents a well-defined class of inputs.

Let us consider the example provided in Figure 1 where a function `nzsum` is meant to check whether for two given 8-bit unsigned integers `x` and `y` their sum is different from zero, for instance to be used safely in a division. Symbolic execution can catch subtle errors and bugs that happen unsurprisingly often in the development of software: the proposed code fragment naively checks whether either argument is non-zero, assuming it as a sufficient condition for their sum to be non-zero as well. However the developer did not consider integer overflows that can happen when the low-level representation cannot hold some values that are possible at run time. While in the C specification these errors lead normally to undefined behavior, the binary code generated for it by a compiler is likely to wrap around the upper limit of the data type. In this example, using two numbers that would 'sum' to `256` (e.g., `120` and `136`) on a larger destination would give `0` in the `uint8_t` representation used to hold the result.

Figure 1 also reports the *symbolic execution tree* that a symbolic execution engine would explore for the considered function. This representation naturally describes the inner working of a generic engine. If we associate each node with an explored instruction, a node can have more than one children when a conditional statement is met. The engine analyzes the condition evaluated by such an instruction and explores two possible alternate paths: one where the condition is met and one where it is not. In the symbolic jargon the executor is said to *fork* two different states, then proceeds by exploring both. At each step the engine maintains a state made of:

- the current assignments of values to variables and memory, which make the so-called *symbolic store* σ;
- the sequence of branching decisions taken along the current path, captured by logical formulas that make the *path constraints* π;
- the current instruction ρ that the engine is about to execute.

When the value of a variable i cannot be determined to a unique value by static code analysis (for instance because it is a program input or an input-derived computation) the symbolic engine assigns it with a symbol α_i. A symbol can initially assume any value allowed by the underlying data type representation, while path constraints refine it as the execution advances. When a symbolic execution of the proposed example begins, we can see an initial state where σ contains two symbols α_x and α_y to represent the input arguments for the function, ρ points to line 1, and π is empty. For the sake of



```
int x = input();
for (int i = 0;
        i < x; i++) {
    // loop body
}
```

Figure 2: Path explosion due to a symbolic loop.

presentation states are labeled with a letter, starting with *A* for the initial one.

As in state A the engine is about to perform an assignment to variable `safe` with value `0` no new symbol is required, and the exploration advances to line 2. State B would then see a symbolic store σ updated with a concrete assignment of `0` to `safe` (we use `s` for short in the tree). Observe that line 2 encodes a branching decision on the value of `x`, which is currently captured by α_x. The engine thus forks two states to reflect the possible outcomes: state C sees π updated with a path constraint for condition $\alpha_x > 0$ being satisfied, while state D sees the negated condition[1]added to its π. From now on both states will capture different classes of inputs for the original variable `x`.

The engine can ideally explore C and D in parallel, or in a practical implementation proceed according to a path scheduling strategy . For state C the engine assigns `safe` with 1 and advances to line 6 in state E where the `assert` instruction is executed. From there it eventually reaches state H that is also a leaf as it corresponds to a `return` statement. From state D the engine considers two alternate worlds: state F where $\alpha_y > 0$ is added to π and `safe` becomes 1, and state G where $\alpha_y = 0$ is chosen instead. State F leads to state I where the `assert` statement at line 6 is traversed without errors, eventually reaching the `return` statement in state L. State G fails instead the `assert` statement since `safe` is `0`, and the engine aborts the execution path.

A user of a symbolic engine can query it to see if at a given set of program states—say the leaves H and L that correspond to `return` instructions—a program property, e.g., $\texttt{x} + \texttt{y} > 0$ holds for any inputs that the corresponding paths can take. Note that the engine is aware of the architectural implications from having both variables represented using 8-bit locations during compilation.

While the desired property always holds for state I, the engine can produce counterexamples for an x86 architecture for state L, e.g., `x = 120` and `y = 136`. To reason over inputs a symbolic engine typically relies on an SMT (Satisfiability Modulo Theories) solver: such solvers can answer questions about logical predicates on the program state as they can take complex logical formulas involving several theories (e.g., integers, reals, bit vectors). SMT solvers can enumerate concrete assignments for the symbolic values that satisfy a predicate for the current π. Symbolic executors also invoke SMT solvers to see for a given branching condition if both outcomes are feasible, in order to rule out unfeasible executions.

Challenges. The symbolic exploration depicted in Figure 1 covers a rather simple example. While from a theoretical point of view symbolic execution is both complete and sound for any decidable analysis, several factors undermine its effectiveness in practical scenarios. We can name three frequently occurring main challenges: (a) path explosion, (b) hard-to-solve constraints, and (c) inaccurate modeling of the execution environment.

Figure 2 shows an exemplifying scenario for *path explosion*. The exploration of the code fragment results in a *symbolic loop* since the number of iterations is potentially unbounded: as the value of symbolic input `x` is initially unconstrained (i.e., its value or bounds are unknown a priori), the engine forks at each iteration yielding an extremely large number of states (up to $2^{31}+1$ in the worst case).

Real-world programs may contain several variants of this scheme, making exhaustive symbolic explorations unfeasible in practice. For this reason users typically define a budget of time and memory for a symbolic exploration, and choose a (oftentimes domain-specific) scheduling strategy for paths to prioritize the exploration of states likely to be more *interesting* for the user's goal, hoping answers (e.g., counterexamples) might be obtained within the chosen budget.

Hard-to-solve constraints are also likely to be encountered when analyzing complex programs: in the presence of certain code patterns, the complexity of constraints accumulated in π becomes hard enough for an SMT solver that query times increase appreciably. For instance, in programs implementing cryptographic or hashing functions a solver may be very slow or even unable to terminate its reasoning on the satisfiability of constraints. SMT solvers incur in fact difficulties when analyzing non-linear operations, and for certain theories the decision problem is undecidable in general, leaving heuristics as the only hope for looking up satisfying value assignments.

Another crucial and more frequent source for hard constraints is represented by memory accesses to a symbolic destination. Symbolic executors may in fact generate long and complex constraints to encode the uncertainty over memory operations involving input-dependent pointers that are symbolic. A common strategy in this case is to perform *concretization*: the engine restricts the set of values that a symbolic input may assume typically to a single concrete value, trading better scalability for the possibility of missing interesting behaviors and states in the exploration.

Finally, *environment modeling* is another compelling issue in symbolic execution. Reasoning over the possible outcomes of the program interactions with the operating system and other components of the software stack (e.g., libraries) is all but trivial. As symbolically executing library and kernel code would lead to hardly governable path explosion phenomena, executors often implement *models* that succinctly capture the effects over a symbolic state of executing a library or system call. However, models are hard to develop and easy to get wrong [9]. Also, they often perform concretizations that can lead an executor to miss interesting states.

Applications. Despite the above-mentioned scalability challenges that symbolic execution has to face, this program analysis has seen an extensive usage and adoption in several application scenarios over the last three decades. Introduced as a technique for generating test cases [25,29], it has been used in several research and industrial projects targeted at detecting software bugs [12,15,20].

In a 2012 journal article [20] Microsoft provided one compelling example of its effectiveness in software testing, revealing how their symbolic executors (which have been running 24/7 in the testing process of many Microsoft products since 2008) found about one third of the bugs related to file inputs when developing Windows 7.

Symbolic execution in the last few years has also gained in popularity in the software security domain. In particular, two of the most prominent applications have been vulnerability detection [36,40] and bug exploitation [5,14]. Recently symbolic execution has also proved to be valuable in reverse engineering tasks over malicious software [8] and their communication protocols [10]. Remarkable evidence of the importance of this technique in the software security community was observed in 2016 during the DARPA Cyber Grand Challenge where symbolic execution was at the core of most of the cyber reasoning systems participating in the competition [37]. The systems ran however in the DECREE environment, a stripped-down software stack that is a strong testing bed for binary reverse engineering, exploitation, and patching, but not very representative of the complexity and entanglements from a realistic fully fledged software stack. Due to space limitations, we refer the interested reader to recent surveys [9,35] for further uses in software security.

3 Related Work

Visualization techniques have helped in several research problems from the software visualization and code execution analysis domains, e.g., static and dynamic code analysis [6,18,21], trace analysis and results explanation [7], and performance analysis [2], being some of them tailored to the cyber-security domain (e.g. analysis of malware [33], static analysis of vulnerabilities [4], writing ROP exploits [3]). However only a few works [13,22–24] have investigated visual solutions when performing symbolic explorations.

`vsdb` [22] and its follow-up SED (Symbolic Execution Debugger) [23] aim at supporting users in debugging sessions over programs available in source form (Java). Both tools build on the observation that a symbolic exploration can start from an arbitrary program point, treating as symbolic any memory contents that would be otherwise available if execution had started from the standard entry point. The tools support the insertion of breakpoints in the symbolic exploration and can visualize the symbolic execution tree using nodes containing source code lines. Additionally, they can produce concrete values for the symbolic store of one or more paths and show the resulting heap configurations [28].

Two added features make SED more appealing to users. Firstly, the use of program slicing [41] to highlight which statements contributed to the value hold by a variable in a given symbolic state, making it easier for a user to understand the dynamics of a bug. Secondly, it allows users to provide specifications over functions and basic blocks for two uses. Specifications can help the symbolic engine skip portions of code that may lead to path explosion, but also encode properties that users may want to verify over the states encountered during the exploration. Although our work shares several traits with `vsdb` and SED, our application context is different: we target symbolic exploration of binary code, which can be more challenging than source-level analysis for a variety of reasons [9], and consider larger and more complex programs in our experiments when dealing with path explosion. As we will detail in the next sections, we propose alternative representations of the symbolic execution tree and visually encode several exploration features into the CFG (Control Flow Graph) for the code.

The Symbolic Execution Visualizer (SEViz) [24] targets Pex, a symbolic execution-based test generation tool from Microsoft for .NET code. After generating new test inputs with Pex, SEViz can

[1] In the domain of unsigned numbers $\alpha_x \leq 0$ simply becomes $\alpha_x = 0$.

visualize the symbolic execution tree encoding several kinds of information through the shape, border, and color of the nodes. Examples include using an ellipse to highlight SMT solver invocations to reason on the node, a double border to indicate that a corresponding source line was found, while a red-colored leaf indicates that the path ended with an exception or other error. Unlike our work, SEViz does not address the problem of keeping the representation of the symbolic tree compact, which may not be ideal in large explorations.

To overcome this limitation [13] proposes the PexViz viewer, which takes the extreme approach of visualizing path explorations through a Variant Control Flow Graph (VCFG) that displays only every branching condition and the entry point as nodes. Although more compact, a VCFG hides most program instructions as it hinges on the implicit assumption that the user has a good knowledge of the program semantics at each step. However, this is not typically the case with the domains we target, and can be difficult in general over binary code due to the loss of high-level information during compilation. A more crucial difference is that SEViz and PexViz do not provide users with means to affect the exploration, as their intended usage is typically as inspection tools.

Of a different flavor is Derailer [32]: for security assessments of Web applications, it uses symbolic execution to identify when and how their data gets exposed. The user inspects and evaluates the identified conditions for security concerns, then Derailer points out missing security checks in sensitive areas. [32] pursues different goals than ours: symbolic execution is the key to obtain exposure data involving specific APIs, but scalability and other concerns typical of a general usage are out of scope. Yet we mention it as one of the few cases where users deal with data (indirectly) derived from a symbolic exploration.

4 The Visual Analytics solution

Scenario. In order to provide the reader with some context on what motivated our work, in the following we describe common usage patterns for security researchers when using symbolic executors. A subject of interest for an exploration could be the verification of some program property as in the introductory example of Section 2, but also the identification of inputs that let a software reach specific program points as in typical code analysis and reverse engineering activities.

The user would typically set up the engine by choosing some heuristics first: those can prioritize paths during the search (using general-purpose strategies like BFS, DFS, and random search, or also domain-specific ones), control the concretization process of symbolic values when deemed necessary, and model specific aspects of the surrounding software environment (e.g., specifying API models). The user would then choose a resource budget for the exploration, made of a temporal duration and a maximum memory occupancy, as forking and maintaining new states increases the footprint of the engine. Choosing a budget is necessary due to likelihood of incurring path explosion problems.

When the user begins the experiment, there is typically no interaction with the symbolic engine. The latter is used essentially as a black-box system that produces some statistics and log files for the explored paths. Occasionally the user may be lucky enough to get counterexamples or satisfying inputs on a first attempt, but in practice it is more likely that the exploration has to start over with different choices of heuristics and low-level parameters. Understanding what needs to be changed is left to the user, but this can be non trivial as the user has very limited information on what happens internally in the engine.

Our visual analytics solution aims at providing the user with actionable information for figuring several details from an exploration.

Architecture. Our system targets the analysis of programs available in binary form, the natural target of many security research applications, but with some adaptations it could also apply to symbolic executors that work on source code. A back-end component is coupled with angr [37]—a very popular engine among security experts—for the analysis of the symbolic states and uses JSON to exchange data, while the front-end is agnostic to the underlying engine and builds on D3.js [11] and dagre [1] as technologies.

Usage. Our solution assists users in visualizing characteristics and distribution of program states, and lets them alter parameters of the exploration either to affect the continuation of the current analysis, or to start a new one when major changes are required.

The first mode is useful when the user believes that live tuning of the current exploration settings could help the engine reach the desired goal, avoiding the repetition of analysis work otherwise needed for a new exploration to reach the current point with the updated settings. The second mode constraints a new exploration based on insights obtained under the current settings, useful for instance when the user realizes that those have led the engine to overlook possibly interesting states.

The visualizations in our prototype encode three main aspects of an execution that in turn can yield actionable insights to the user:

- program locations that see (intense) state forking activity, which typically leads to rapid exhaustion of the budget;
- symbolic data involved in branching decisions, useful to figure out the dynamics behind state forking activities;
- program inputs and code points where symbolic data originates.

By actionable insights we mean that we support the user in refining the exploration in two ways:

- controlling control flow graph (CFG in what follows) edges and blocks that the executor traverses, with different policies;
- placing constraints in the symbolic data generation process, either to refine the admissible ranges that the engine shall consider, or to drive concretization policies for hard-to-solve constraints.

As we will detail throughout Section 4.2, these actions update our visualizations by presenting how the symbolic exploration would be affected had the user-specified criteria been put in place from the beginning. Changes can affect two dimensions: symbolic tree nodes that would still be part of the exploration (we present the subtree that matches the desired criteria) and effects on code coverage (we update the visualizations associated with the code under analysis).

For function identification we use the analysis and disassembler components of radare2, but other reverse engineering frameworks could be used as well. CFG nodes embody a standard definition of basic block: a sequence of instructions with a single point of entry and that ends with a control transfer instruction. CFG edges encode such transfers.

4.1 Back-end Component

The back-end component of our visual analytics solution deals with two main aspects: i) extracting the internal state of the underlying symbolic execution engine; and ii) controlling the exploration, e.g., applying rules—for the continuation of the current exploration or a new one—that reflects the settings specified by the user in the front-end.

Concerning the internal state of the symbolic execution engine, the component collects:

- the CFG of the code under analysis;
- the nodes of the symbolic tree;
- the source and nature of symbolic data (e.g., file access, command-line argument, etc.);
- locations in which symbolic data is used for branching decisions;

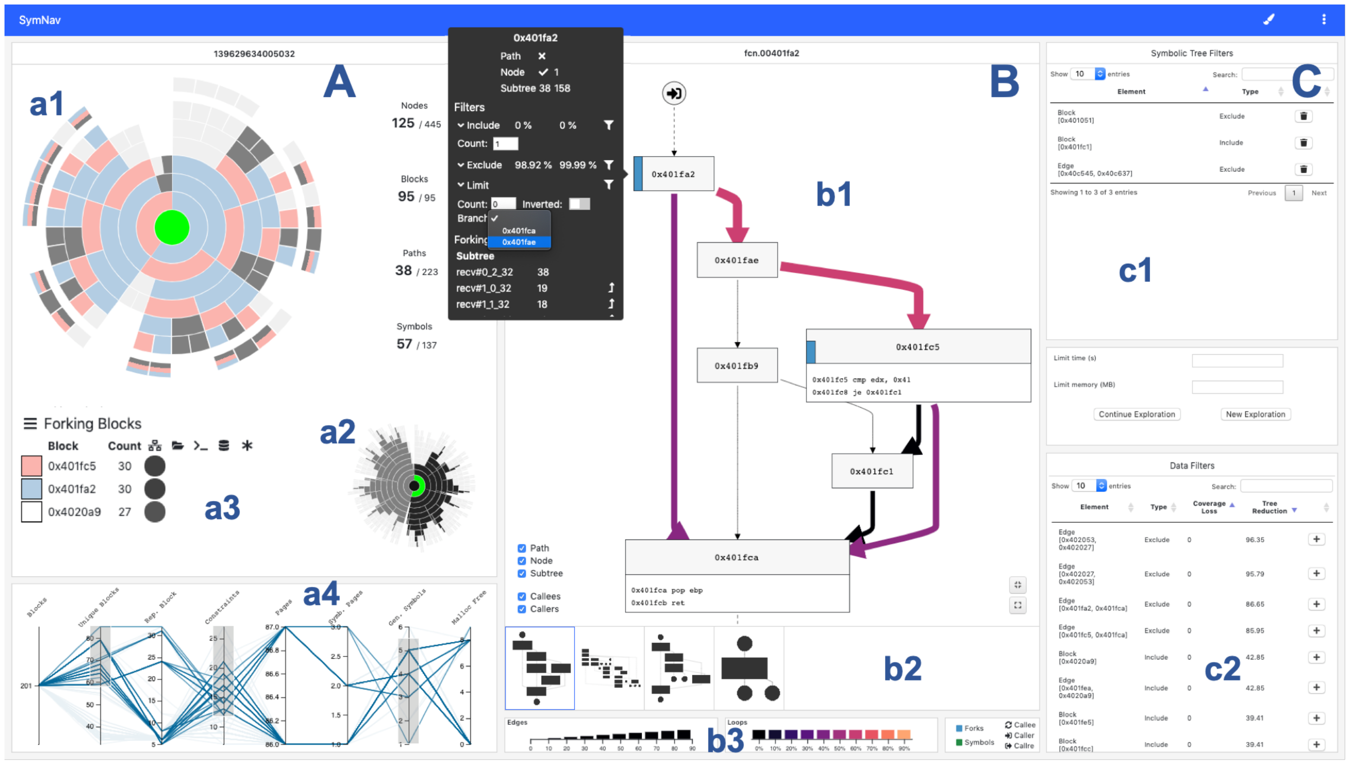


Figure 3: Overview of SymNav visual component. **(A)** The *Symbolic Tree Environment* containing: **(a1)** the *Focus Tree*, **(a2)** the *Context Tree*, **(a3)** the interactive table for tree encoding and **(a4)** the *Parallel Coordinates* visualization of paths. **(B)** The *Control Flow Graph Environment* representing the control flow graph of the function under analysis divided in: **(b1)** the main view, **(b2)** the timeline and **(b3)** the legend of the encoding. **(C)** The *Steering Environment* containing: **(c1)** the *Symbolic Tree Filters Pane* and **(c2)** the *Data Filters Pane*.

The component computes two kinds of analytical information: i) cumulative statistics for an exploration path, including (but not limited to) the number of visited CFG nodes, the number of constraints in π, and the number of memory pages in use; and ii) prediction on coverage loss and tree size reduction when pruning paths using exploration control features. The latter may provide actionable information to the user and is available for every edge and node visited in the current exploration.

Concerning the exploration control, the component allows for filtering paths, in accordance with how these traverse the CFG or refining input ranges; such filtering activities can be used for either resuming the exploration or starting a new one. Refinement capabilities based on CFG nodes, edges, and topology are meant to meet common execution patterns from programs. Consider for instance loops that process formatted input fields: those could likely yield many uninteresting paths for ill-formed data, but enforcing a certain number of branch outcomes can curtail the exploration to avoid them. Putting a minimum or maximum threshold on branch traversal may instead be useful when processing long strings that contain heading or trailing patterns that are important for the program semantics. In more detail, we have implemented exploration control means for:

- starting an exploration providing an angr project specification (including, e.g., starting point for exploration, pre-constraints on inputs, or custom API models) and a budget (duration and memory occupancy) or resuming the current one with a new budget;
- specifying exclusion conditions for paths, e.g.:
  - given a CFG node n (or edge e) and a constant k, it is possible to drop any path in the symbolic tree that traverses n (or e) fewer than k times, or that does not traverses it at least k times.
  - given a CFG branching node n, and a constant k, and an outcome b for the branch (taken or not), keep tree paths that explore n where since the first traversal of n, either $\neg b$ is never taken or it is after b was taken at least k times in a row from the first traversal.
  - given a CFG branching node n, and a constant k, and an outcome b for the branch (taken or not), keep a tree path that explores n either if it traverses it at most $k-1$ times, or from the k-th time on b is the only outcome observed for the branch.
  - further refine the symbolic tree such that by restricting the value of a symbolic data element to a specific range, paths violating such condition(s) are dropped.

When starting over or continuing an exploration, filters are converted into constraints over symbolic data and/or hints to search heuristics; filters referring to dynamically generated symbolic data (e.g., network packets) are not considered when starting a new exploration because such dynamic data is not uniquely identifiable across different explorations. We observe, however, that this is a recurrent trait in the symbolic execution practice.

4.2 Visual Component

The SymNav visual component (Figure 3) allows for analyzing the symbolic tree and driving further computations. The visual solution has been designed together with four symbolic execution experts following a user-centered design paradigm [31]. During the first meeting, the experts described the typical workflow and outlined the initial user requirements. Four meetings (lasting two hours) followed in which we proposed different data sketches to validate the design choices and refine the requirements.

Given the challenges exposed in Section 2, SymNav is composed of three interactive and coordinated environments, each of them tailored to a specific analysis workflow and aspects of the symbolic execution: the exploration of the symbolic execution tree (*Symbolic Tree Environment*), the analysis of the CFG (*Control Flow Graph Environment*), and the steering of possible analyses on the previous two environments (*Steering Environment*).

4.2.1 Symbolic Tree Environment

The *Symbolic Tree Environment* (Figure 3.A) copes with the exploration of the symbolic execution tree and was the first designed part of SymNav. Requirements pointed out the need of: a) showing the hierarchical structure of the symbolic execution tree in a compact way, highlighting points where symbolic data are used and generated, b) navigating the symbolic tree preserving the context, and c) analyzing and pruning the tree according to the characteristics of the paths.

Due to the high number of states that usually compose the symbolic tree and to the particular focus of the users on states that fork two (or more, as it may happen with indirect jumps) states, we have decided to compact the structure of the tree through the aggregation of the sequences of states without forks (i.e., states with one child) in single nodes , e.g., the nodes C, E and H of the tree presented in Figure 1.

The *Symbolic Tree Environment* shows the symbolic execution tree along different coordinates, as visible in Figure 3.A. The high number of nodes pushed us to investigate space-filling techniques [39], exposing the users both to the treemap and the sunburst methods, and choosing the latter due to user preferences for sunburst hierarchy representation and traversal ("it better conveys the hierarchy") and to the possibility of adding additional information on leaves. In order to better support the tree exploration, we designed this environment using the focus+context paradigm [30]. The *Focus Tree* (Figure 3.a1) shows the symbolic exploration tree: a sunburst encodes the first n levels of nodes that are surrounded by a ring whose elements represent the subtrees from the n-th level nodes (if they are not leaves). Selecting (clicking) a node updates the view showing the subtree of the selected node, with it higlighted in green at the center of the new sunburst. This view is paired with the *Context Tree* (Figure 3.a2) that represents the whole tree in a classic sunburst, highlighting the selected node (still in green), the path from the root to the selected node (traversed path) and its subtree.

For each node of the *Focus Tree* the color encodes, according to the user choice, information about either its basic blocks or symbols.

In block mode, the color encodes the basic block that causes the fork; an interactive table (Figure 3.a3) below the sunburst shows all the basic blocks that cause at least one fork in the current tree; blocks are sorted in descending order with respect to the number of forks. A fork is associated with one or more symbols that are classified according to their origin (i.e., network, file system, command line, memory, others). These symbols classes are organized in a matrix aligned with the table, and the shade of the cell color encodes the number of symbols of the class involved in the forks on the basic block. Due to the possible high number of "forking blocks", the user can select which basic blocks to show on the tree picking them from the interactive table, with a dynamic color assignment that will show their distribution on the *Focus Tree* (by default, the top-k are selected). While a node of the tree has a single basic block responsible for the fork and its encoding is quite straightforward according to this paradigm, the encoding of the ring elements requires special attention because each element represents a collection of nodes. If all the nodes of the subtree fork due to the same basic block, the element ring is encoded with the color of that block; conversely, up to k colors encode the top-k list of basic blocks in the subtree sorted in descending order with respect to the number of forks.

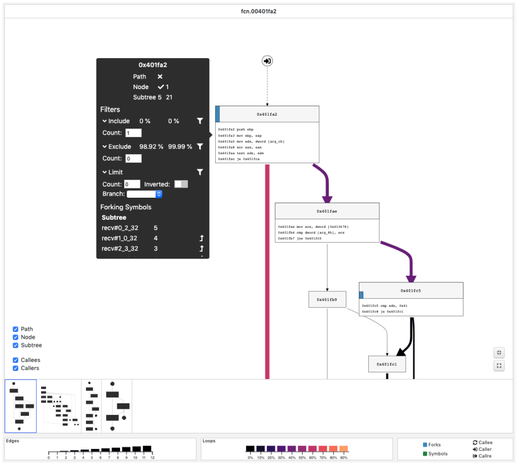
The symbol mode is similar to the block mode but it focuses on symbols; the interactive table lists all the symbols involved in at least one fork, sorted in descending order with respect to the number of forks, giving information about their class and the generation depth. Being the number of symbols involved in a single fork ≥ 1 all nodes are encoded with up to k colors corresponding to the top-k list of symbols sorted in ascending order with respect to the generation depth.

These visualizations show the topological structure of the tree, but the design process pointed out the need to explicitly characterize the paths that traverse the tree in terms of their characteristics (e.g., length, number of symbols, number of memory pages, etc). Due to the multidimensional structure of this information we have encoded paths using the *Parallel Coordinates* visualization [26] (see Figure 3.a4, below the *Symbolic Tree Environment*). This visualization helps both in identifying relationships and patterns among dimensions and in dropping out parts of the tree that do not meet the user's needs. When Brushing on the axes of the parallel coordinates the tree is pruned accordingly, with reduced color opacity for dropped parts on both *Focus Tree* and *Context Tree*. Finally, a set of descriptive statistics are presented in the right part of the environment, listing indicators (selected / total) about number of nodes, basic blocks, paths and symbols, updated in accordance with user selections.

4.2.2 Control Flow Graph Environment

While the *Symbolic Tree Environment* is well suited to explore in different ways the symbolic execution tree, a real understanding of the execution paths requires the analysis of the underlying code. Due to this reason, we have designed the *Control Flow Graph Environment* (Figure 3.B) to represent the control flow graph of the inspected functions, enriching it with information grasped from the symbolic execution engine (i.e., information about the currently selected nodes, the traversed path, the subtree, or a combination of them).

Regarding the CFG, we have evaluated nodes-links and matrix-based representations, choosing the former in light of a) the need to display complex information at node level, b) the need to support tasks like following path(s) (see, e.g., [19]), c) evidences collected from related proposals (see, e.g., [18]) and d) the clear preferences of our users, collected during project meetings comparing proposed matrix-based and nodes-links based representations of the same CFG.



0x401fa2
Path ✕
Node ✓ 1
Subtree 5 21
Filters
Include 0 % 0 %
Count: 1
Exclude 98.92 % 99.99 %
Count: 0
Limit
Count: 0 Inverted:
Branch:
Forking Symbols
Subtree

recv#0_2_32	5
recv#1_0_32	4
recv#2_3_32	3

```
0x401fa2 push ebp
0x401fa3 mov ebp, esp
0x401fa5 mov edx, dword [arg_ch]
0x401fa8 xor eax, eax
0x401faa test edx, edx
0x401fac je 0x401fca
```

```
0x401fae mov ecx, dword [0x410678]
0x401fb4 cmp dword [arg_8h], ecx
0x401fb7 jne 0x401fc5
```

```
0x401fc5 cmp edx, 0x41
0x401fc8 je 0x401fc1
```

Figure 4: *Control Flow Graph Environment*. The tooltip shows additional information on paths, symbols, and constraints. Paths that cover an edge numerous times define its brightness: the edge on the left is traversed 21 times by 5 paths, this explains why it is particularly bright.

We represent basic blocks (identified by their memory address) using rectangles, while solid directed edges encode their connections. To better support the navigation of the different functions, the user can choose to see the functions that were called by the current one (*callees*) or the ones that call it (*callers*). These functions are represented as small circular icons that recall their roles. Callees are linked to the basic block that calls them through a dotted edge; callers were initially linked both to the called basic block and to the returning basic block, but this encoding confused the users, which identified it as a loop. After some iterations, we chose to represent a caller as two distinct circles—one connected to the called basic block and one to the basic block it returns to—because according to the users this encoding better represents the execution flow. Selecting a function reveals its CFG in the main view. A timeline (Figure 3.b2) is provided to navigate the temporal sequence of inspected functions; each function is represented with a thumbnail of its CFG, allowing the user to move back and forth among different portions of the inspected code.

According to the users' needs to relate inspected basic blocks to the symbolic execution, we identify two kinds of information that should be encoded on the basic blocks: their position in the current tree and their use of symbols. The border color of the basic block encodes the position: green indicates that the block is in the currently selected node in the tree, black that it is in the traversed path or in the subtree (if at least one of these levels is selected). Moreover, by mouse-hovering on a basic block all the nodes that contain it are highlighted in the *Symbolic Tree Environment*.

We distinguish between the symbols that a basic block uses in a fork and the symbols that it generates. In particular, a vertical blue bar on the left of the basic block indicates that the block forks, being the height of the bar proportional to the number of symbol occurrences in the forks. Similarly, a vertical green bar on the right of the basic block indicates that the block generates symbols, with its height proportional to the number of generated symbols. These pieces of information are quite relevant because they are indicators of basic blocks that produce a high number of paths (symbol occurrences in forks) or that present many input operations (generated symbols). Finally, by clicking on a basic block the user can expand it and inspect the contained instructions.

We use the edge visual encoding to convey information about the symbolic execution: as visible in Figure 4, the thickness of an edge encodes how many distinct execution paths pass through it, while the brightness of the edge color encodes how many times execution paths pass through it; this design choice effectively highlights edges involved in loops that are very often a target of the user analysis because they can consume a high portion of the resource budget. An informative legend (Figure 3.b3) about the visual encoding of edges (thickness), loops (magma color-scale), forks (dark-blue) and symbols (dark-green) is present in the bottom part of the environment.

Mouse-hovering the elements of the graph triggers a tooltip with all the associated information: for a basic block, the tooltip shows information about its occurrences in the portion of the symbolic tree under analysis, and provides the list of the symbols that it generates and those it uses in the forks; for a symbol, the node in which it has been generated is highlighted in the *Symbolic Tree Environment* and clicking on it causes the node selection.

Additionally, from the tooltip the user can directly exclude a basic block from execution (black-listing), necessarily include it in the execution (white-listing), or ask for a fixed number of basic block executions. These features are very important when the user needs to manage the resource budget and, if well administered, can help in raising the coverage of the symbolic tree. For this reason, the tooltip shows also the predicted coverage loss and tree size reduction from applying the filter.

Finally, in the bottom-right corner two shortcuts are provided to expand/collapse all the basic blocks.

4.2.3 Steering Environment

The *Steering Environment* (Figure 3.C) has two main goals: presenting in a homogeneous way the choices made during the analysis by the user, and providing suggestions for continuing the analysis in order to support the steering of the symbolic exploration. This environment is composed by two panes.

The *Data Filters Pane* (Figure 3.c2) is designed to guide the user, suggesting possible elements of interest to steer the exploration. It analyzes the symbolic trees and provides in an interactive table the list of its blocks/edges associated with the suggested exclusion/inclusion conditions and predictions about coverage loss and symbolic tree size reduction. The list is sorted in descending order for coverage loss (the user aims at maximum coverage) and ascending order for tree size reduction. In this way the user can steer the symbolic exploration matching the presented information with the desired degree of coverage and resource budget allocation, obtaining a form of guidance in steering the symbolic exploration opposed to a free exploration.

This activity is supported also by the *Symbolic Tree Filters Pane* (Figure 3.c1). It lists the filters currently applied on the symbolic tree during the analysis in the *Control Flow Graph Environment*, and presents them in a unified and more homogeneous way as an interactive table for both basic blocks and edges (normally these constraints are scattered in different functions). It also provides the means for defining the resource budget (duration time and memory occupancy), and triggering a new exploration (new exploration button) or resuming the current one (continue exploration button). In both cases it is possible to apply the current active set of filters to the exploration. In this way the user can manage the trade-off between fine-tuning of a symbolic exploration and its resumption (or starting a new one), having an easy way to apply a constructed set of filters to an exploration as a base from where to start again the fine-tuning process. Finally, mouse-hovering on an element in the two panes highlights it in the rest of the system.

5 Usage Scenarios

This section presents two usage scenarios [27, 38], showing how SymNav assists a human in making sensible decisions when applying symbolic execution techniques to the domains of malware analysis and vulnerability detection. In both scenarios we use BFS as search heuristic for the exploration, which is also the default one in angr. The first usage scenario we outline can be followed in the provided supplemental video.

Netwire. Netwire is a Remote Access Trojan (RAT) malware family sold on the black market that exhibits a variety of malicious behaviors, such as logging keystrokes and allowing the attacker to remotely execute commands on the infected machine. Although Netwire was employed by cybercriminals to steal payment card data as recently as in 2016, its variants could be found in the black market since 2012. The one we considered[2] exposes 51 different commands implementing different functionalities, such as sending a file to the victim's computer, taking a screenshot, and stealing browser credentials.

Due to the large number of commands, it is not trivial for a malware analyst to understand the exact sequence of instructions that will lead to the execution of a specific command. In particular, since the various behaviors of Netwire are triggered by TCP packets, an analyst is interested in figuring their structure, contents, and possible dependencies (e.g., if an authentication phase with a remote counterpart is required).

Symbolic execution has proven useful to reconstruct valid input packets in such scenarios before [8]. However, due to path explosion the exploration does not reveal the commands from this sample in a reasonable time and memory budget, unless some hints are provided

[2]MD5 hash: `37e922093d8a837b250e72cc87a664cd`.

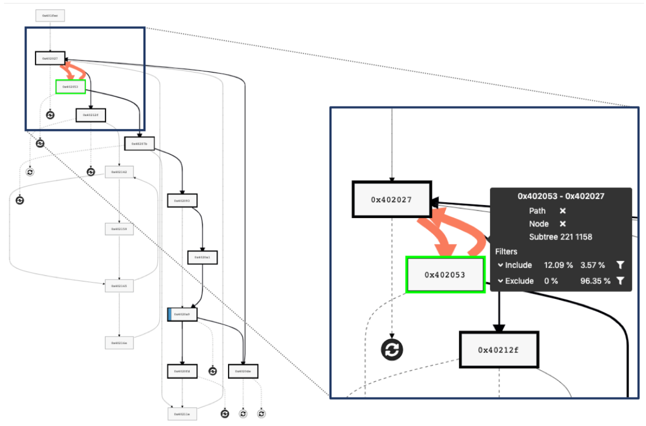
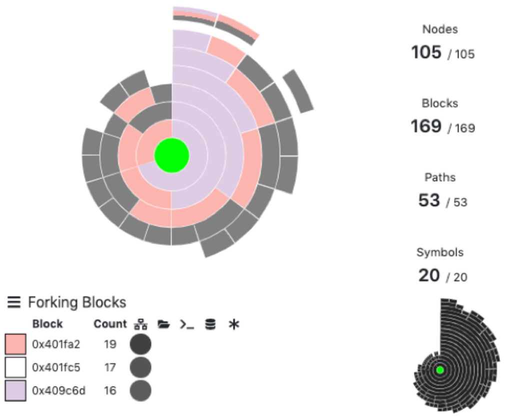


Figure 5: Netwire's receive loop. The malware inspects the packet and retries the receive if it is ill-formed.



≡ Forking Blocks

Block	Count
🟥 0x401fa2	19
⬜ 0x401fc5	17
🟪 0x409c6d	16

Figure 6: Netwire's *Focus Tree*. The spiral-like shape of the tree suggests the presence of a loop. Indeed, the malware looks up patterns in the received packet one byte at a time.

manually. In particular, in our experiments we observed that if we run angr with no manual assistance for a budget of 12 hours and 64 GB of RAM, it exhausts memory by generating ~10,000 states that do not execute any of Netwire's commands. SymNav can help the analyst steer the exploration in order to extract the sequence of packets triggering each command without requiring deep prior knowledge of the sample.

Let us now outline the possible steps an analyst could follow when using SymNav. Every filtering step is based on the data collected by the tool's back-end, after symbolically exploring the analyzed program. We proceed in small incremental steps; we give the back-end a time budget of 5 minutes for each symbolic exploration and we manage to reach every command by spending a total of 20 minutes.

1. We start by selecting the root of the *Focus Tree*. As a result, we can observe in the *CFG Environment* the function in which the first state fork takes place. If we navigate to its caller, we can see that its call site is in a loop: this is easily noticeable as the inbound and outbound edges of the basic block for the call are visually thick and bright (*Figure* 5). If we examine the code, we can see that it encodes a packet processing logic that advances only upon collecting value `0x41` as header of a command sequence. If we move the mouse pointer over the loop's back edge, we can see that the coverage loss from excluding paths going through this edge from the symbolic tree would be zero, with a tree size reduction of 38.18%. This suggests that excluding it could be beneficial: actually this operation would remove from the tree all the paths trapped in the loop due to ill-formed packets with no derived coverage loss. Therefore, we proceed with this operation and let the back-end resume the symbolic exploration on the pruned tree.
2. On expiry of the 5-minute budget the front-end displays the updated symbolic tree. When selecting its root, the *CFG Environment* shows as fork point a loop that looks up patterns in the received packet one byte at a time. The fact that we are dealing with a loop construct that naturally leads to nested forking is also reflected by the spiral-like shape of the *Focus Tree* (*Figure* 6). We then notice that the loop generates two clearly distinguishable groups of paths for the same blocks and edges; by employing the caller icons to explore where such paths originate, we reach a function containing numerous basic blocks. Taking a closer look at its code, we realize that this function is the command dispatcher, where Netwire parses the received packets and executes the corresponding commands.

   From the *CFG Environment* we can see that the two path groups lead to the execution of the same command, leaving the other 50 unexecuted (*Figure* 7). From an analyst's experience, this could indicate that an authentication phase is required. Furthermore, we notice that paths from one group traverse a large basic block containing many calls to interesting APIs (e.g., `GetUserName`, `GetHostName`, etc.), while for the other group Netwire closes the Internet socket and exits the function. In light of these findings, we deduce that paths in the first group have likely gone past the authentication phase. Therefore, we can reasonably opt to apply an include filter to the first block traversed only by paths in the first group, and then let the exploration resume.
3. By examining the updated tree, we realize that the root node is now forking again in the first function observed in step 1. Navigating back to its caller, we notice that all the paths leaving the dispatcher go backwards to the function entry block and eventually the receive loop, waiting for the next command (*Figure* 8). From there some paths leave the loop and go back to the dispatcher, but unfortunately the exploration stops due to time budget depletion before executing another command. As a result, we opt to filter the tree by keeping only those paths that traverse the first block in the dispatcher at least twice. We then resume the exploration one more time, finally reaching our end goal: the visualization now helps us identify distinct blocks related to the execution of each command supported by the RAT sample. The inspection of each group of blocks details in fact the structure of every command.

Touch. As a second usage scenario we present the analysis of a command-line Linux utility, looking for memory-related errors that could be a security concern. Touch belongs to the GNU Core Utilities and can modify access and modification dates of files and directories. We consider the 8.21 release, part of several Ubuntu LTS releases until 2015. As we have seen in Section 2, symbolic execution can reason on complex program properties: we thus encoded two predicates that the engine evaluates during the exploration. The first checks whether a symbolic memory pointer can reference multiple dynamically allocated objects: this behavior can be suspicious in some contexts [16], and captures several heap buffer overflows error. The second checks whether an address from a non-allocated area is used as memory operand: this may happen in, e.g., double-free, invalid-free, or use-after-free errors.

Since Touch can take multiple command-line arguments, in the symbolic execution practice it is common to use concrete values for all but one, thus symbolically exploring one option at a time. But even so in our experiments the symbolic engine generated ~37,000 states, exhausting a budget of 12 hours and 64 GB of RAM without revealing potentially interesting paths.

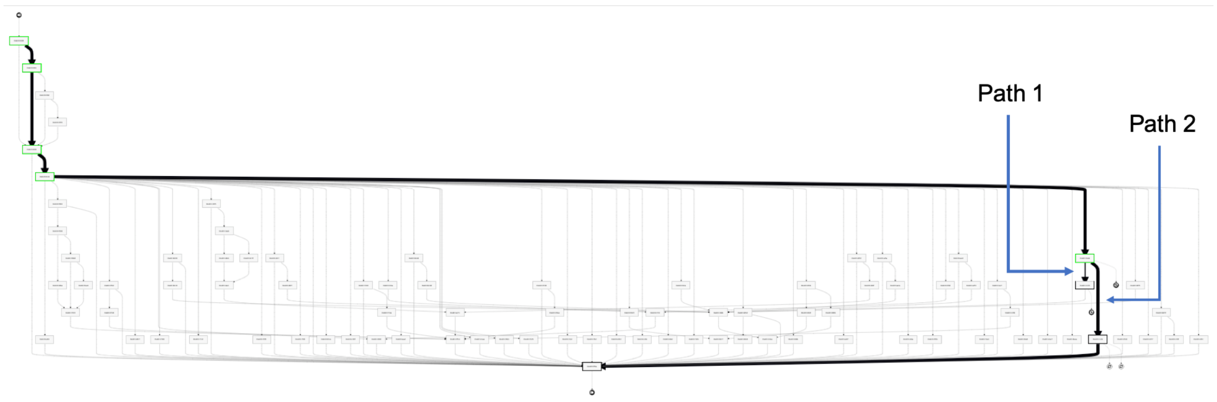


Figure 7: Netwire's command dispatcher. The function executes a different command depending on the value received from the network.

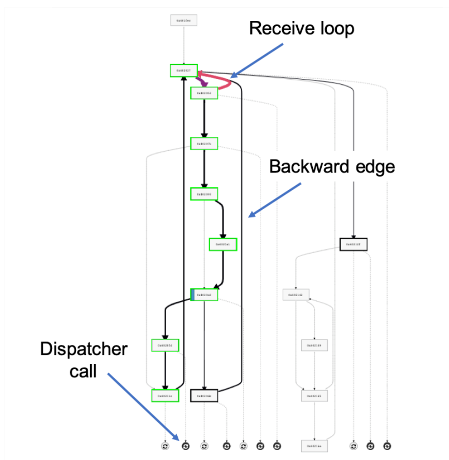


Figure 8: Netwire's main function. The malware receives a packet from the network, executes a command in the dispatcher and, following the backward edge, returns in the receive loop waiting for other commands.

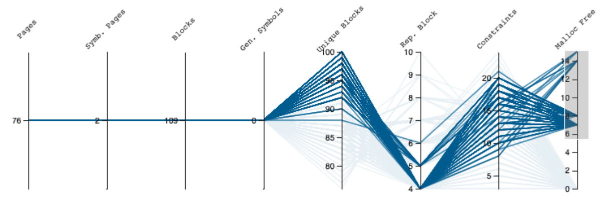


Figure 9: Touch's paths representation and selection using parallel coordinates. We filter out paths without `malloc` or `free` operations.

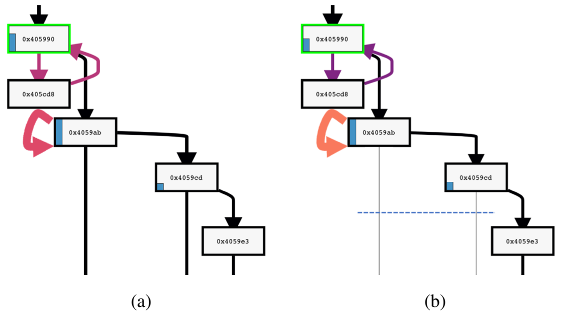


Figure 10: *Touch's CFG Environment*. We show the difference between Touch's CFG before (a) and after (b) the filtering operation made with the parallel coordinates.

We thus attempt an exploration in SymNav with the same settings used for Netwire and after the initial run we can see, by selecting the root of the *Focus Tree*, that the first function to cause a state fork is `parseDateTime`. The function comprises many basic blocks, but only some (those closer to the function's entry point) are covered by the exploration. From the *CFG Environment* the user can also see several loops being explored. As we look for interesting memory behaviors, we can use the parallel coordinates to filter out for instance paths that do not perform any `malloc` or `free` operations (*Figure* 9). This choice proves to be effective, as once the symbolic tree gets updated we can see that the remaining paths are concentrated (and most of them stuck) in a single loop (*Figure* 10). We apply an include filter on the CFG edge that enters the loop.

After another exploration round, we see that one block in such loop gets annotated as a result of the interaction with the back-end. The predicate for operands falling in non-allocated regions was actually met in it along some path, resulting in an invalid-free error. We can ask angr through the back-end to produce a concrete input assignment that exercises the path: the SMT solver produces a string `--date=TZ="""`, where the presence of an extra quote triggers the error. This vulnerability is also known as CVE-2014-9471.

6 Concluding Remarks

In this paper we have presented a novel Visual Analytics solution to assist users in symbolic executions of real-world programs. SymNav provides a compact representation of the symbolic tree and of the CFG of each function enriched with information that helps users identify state forking points and symbolic data responsible for path explosion. SymNav also lets users apply rules that can refine the continuation of the current exploration or drive a new one. We presented two usage scenarios where the computational budget required to generate the paths relevant for the analysis was reduced significantly with respect to standard automatic explorations without SymNav.

A few directions could be explored in future work. One open problem involves the visualization of path constraints: as textual representations would not help when analyzing real-world programs, researchers should explore new ways to encode them compactly, especially if they may appear in possibly distinct large sets of paths. On the practical side, we could support additional symbolic engines: SymNav could reach a broader audience by implementing a back-end for the source-level KLEE [12] executor, or for *concolic* executors based on QEMU [15] or DBI [17, 34] that mix symbolic reasoning with concrete executions. We may extend SymNav to tune low-level aspects of the symbolic exploration, allowing users to select on-the-fly a different search heuristic for path prioritization or specify custom concretization strategies from the front-end.

Acknowledgments

We are indebted to Alessia Palleschi for her help in preparing the demonstration video. We would also like to thank the anonymous VizSec reviewers for their valuable comments.

References

[1] Github - dagrejs/dagre: Directed graph layout for javascript. `https://github.com/dagrejs/dagre`. (Accessed on 06/19/2019).

[2] A. Adamoli and M. Hauswirth. Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pp. 73–82. ACM, New York, NY, USA, 2010. doi: 10.1145/1879211.1879224

[3] M. Angelini, G. Blasilli, P. Borrello, E. Coppa, D. C. D'Elia, S. Ferracci, S. Lenti, and G. Santucci. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, VizSec '18, 2018. doi: 10.1109/VIZSEC.2018.8709204

[4] H. Assal, S. Chiasson, and R. Biddle. Cesar: Visual representation of source code vulnerabilities. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8, Oct 2016. doi: 10.1109/VIZSEC.2016.7739576

[5] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. In *Proceedings Network and Distributed System Security Symposium*, NDSS'11, 2011.

[6] I. Bacher, B. M. Namee, and J. D. Kelleher. On using tree visualisation techniques to support source code comprehension. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 91–95, Oct 2016. doi: 10.1109/VISSOFT.2016.8

[7] I. Bacher, B. M. Namee, and J. D. Kelleher. On using tree visualisation techniques to support source code comprehension. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 91–95, Oct 2016. doi: 10.1109/VISSOFT.2016.8

[8] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu. Assisting Malware Analysis with Symbolic Execution: A Case Study. In S. Dolev and S. Lodha, eds., *Cyber Security Cryptography and Machine Learning*, CSCML '17, pp. 171–188. Springer International Publishing, Cham, 2017. doi: 10.1007/978-3-319-60080-2_12

[9] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computer Surveys*, 51(3):50:1–50:39, 5 2018. doi: 10.1145/3182657

[10] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu. Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution. In S. Dolev and S. Lodha, eds., *Cyber Security Cryptography and Machine Learning*, CSCML '19. Springer International Publishing, 2019. doi: 10.1007/978-3-030-20951-3_12

[11] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec 2011. doi: 10.1109/TVCG.2011.185

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'08, pp. 209–224. USENIX Association, 2008.

[13] J. Cao, A. Astorga, S. Srisakaokul, Z. Wu, X. Liu, X. Xiao, and T. Xie. Visualizing path exploration to assist problem diagnosis for structural test generation. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 323–325, Oct 2018. doi: 10.1109/VLHCC.2018.8506484

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. 2012 IEEE Symp. on Sec. and Privacy*, SP'12, pp. 380–394. IEEE Comp. Society, 2012. doi: 10.1109/SP.2012.31

[15] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):2:1–2:49, 2012. doi: 10.1145/2110356.2110358

[16] E. Coppa, D. C. D'Elia, and C. Demetrescu. Rethinking Pointer Reasoning in Symbolic Execution. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, pp. 613–618. IEEE Press, Piscataway, NJ, USA, 2017. doi: 10.1109/ASE.2017.8115671

[17] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 14th ACM ASIA Conference on Computer and Communications Security*, ASIACCS '19, 2019.

[18] S. Devkota and K. E. Isaacs. CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum*, 37(3):453–464, jun 2018. doi: 10.1111/cgf.13433

[19] M. Ghoniem, J.-D. Fekete, and P. Castagliola. On the readability of graphs using node-link and matrix-based representations: A controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005. doi: 10.1057/palgrave.ivs.9500092

[20] P. Godefroid, M. Y. Levin, and D. A. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, 2012. doi: 10.1145/2090147.2094081

[21] J. R. Goodall, H. Radwan, and L. Halseth. Visual analysis of code security. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, pp. 46–51. ACM, 2010.

[22] R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pp. 143–146. ACM, New York, NY, USA, 2010. doi: 10.1145/1858996.1859022

[23] M. Hentschel, R. Bubel, and R. Hähnle. The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, Mar 2018. doi: 10.1007/s10009-018-0490-9

[24] D. Honfi, A. Voros, and Z. Micskei. Seviz: A tool for visualizing symbolic execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–8, April 2015. doi: 10.1109/ICST.2015.7102631

[25] W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering (TSE)*, 3(4):266–278, 1977. doi: 10.1109/TSE.1977.231144

[26] A. Inselberg. *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. Springer-Verlag, Berlin, Heidelberg, 2009.

[27] T. Isenberg, P. Isenberg, J. Chen, M. Sedlmair, and T. Mller. A systematic review on the practice of evaluating visualization. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2818–2827, Dec 2013. doi: 10.1109/TVCG.2013.126

[28] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pp. 553–568. Springer-Verlag, 2003. doi: 10.1007/3-540-36577-x_40

[29] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. doi: 10.1145/360248.360252

[30] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pp. 401–408. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. doi: 10.1145/223904.223956

[31] S. Mckenna, D. Staheli, and M. Meyer. Unlocking user-centered design methods for building cyber security visualizations. In *2015 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8, Oct 2015. doi: 10.1109/VIZSEC.2015.7312771

[32] J. P. Near and D. Jackson. Derailer: Interactive security analysis for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 587–598. ACM, New York, NY, USA, 2014. doi: 10.1145/2642937.2643012

[33] G. R. Santhanam, B. Holland, S. Kothari, and J. Mathews. Interactive visualization toolbox to detect sophisticated android malware. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pp. 1–8, Oct 2017. doi: 10.1109/VIZSEC.2017.8062197

[34] F. Saudel and J. Salwan. Triton: A dynamic symbolic execution framework. In *Symp. sur la Sécurité des Technologies de l'Information*

et des Communications, SSTIC'15, pp. 31–54, 2015.

[35] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 317–331. IEEE Computer Society, Washington, DC, USA, 2010. doi: 10.1109/SP.2010.26

[36] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symp.*, NDSS'15, 2015. doi: 10.14722/ndss.2015.23294

[37] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symp. on Security and Privacy*, SP'16, pp. 138–157, 2016. doi: 10.1109/SP.2016.17

[38] D. Staheli, T. Yu, R. J. Crouser, S. Damodaran, K. Nam, D. O'Gwynn, S. McKenna, and L. Harrison. Visualization evaluation for cyber security: Trends and future directions. In *Proceedings of the Eleventh Workshop on Visualization for Cyber Security*, VizSec '14, pp. 49–56. ACM, New York, NY, USA, 2014. doi: 10.1145/2671491.2671492

[39] J. Stasko, R. Catrambone, M. Guzdial, and K. McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663 – 694, 2000. doi: 10.1006/ijhc.2000.0420

[40] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23nd Annual Network and Distr. System Sec. Symp.*, NDSS'16, 2016.

[41] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. doi: 10.1109/TSE.1984.5010248