Symbolic Complexity Analysis using Context-preserving Histories

Kasper Luckow*, Rody Kersten*, Corina Păsăreanu*†
*Carnegie Mellon University, Moffett Field, CA, USA
†NASA Ames Research Center, Moffett Field, CA, USA

***Abstract*—We propose a technique based on symbolic execution for analyzing the algorithmic complexity of programs. The technique uses an efficient *guided* analysis to compute bounds on the worst-case complexity (for increasing input sizes) and to generate test values that trigger the worst-case behaviors. The resulting bounds are *fitted* to a function to obtain a prediction of the worst-case program behavior at any input sizes. Comparing these predictions to the programmers' expectations or to theoretical asymptotic bounds can reveal vulnerabilities or confirm that a program behaves as expected.**

To achieve scalability we use *path policies* to guide the symbolic execution towards worst-case paths. The policies are learned from the worst-case results obtained with exhaustive exploration at small input sizes and are applied to guide exploration at larger input sizes, where un-guided exhaustive exploration is no longer possible. To achieve precision we use path policies that take into account the *history* of choices made along the path when deciding which branch to execute next in the program. Furthermore, the history computation is context-preserving, meaning that the decision for each branch depends on the history computed with respect to the enclosing method.

We implemented the technique in the Symbolic PathFinder tool. We show experimentally that it can find vulnerabilities in complex Java programs and can outperform established symbolic techniques.

***Index Terms*—Complexity Analysis; Symbolic Execution; Guided Exploration**

I. Introduction

Understanding the worst-case algorithmic complexity of software systems has many important applications, ranging from compiler optimizations, finding and fixing performance bottlenecks or improving cybersecurity. Cybersecurity in particular has become a critical goal for companies and organizations at every level. Software systems can be vulnerable to algorithmic complexity attacks when a malicious user can easily build a "small" input that causes the system to consume an impractically large amount of resources (time or memory). By exploiting these vulnerabilities an adversary can mount Denial-of-Service (DoS) attacks in order to deny service to the system's benign users, or to otherwise disable the system. Algorithmic complexity vulnerabilities are the consequence of the algorithms used rather than of traditional "software bugs", and consequently traditional software bug hunting tools are of little use to address the problem. Profilers can be used for finding performance bottlenecks in software, however they are inherently limited by the number of test inputs used and the platform on which the profiling was done.

In this paper, we investigate the use of symbolic execution, a systematic program analysis technique [6], [12], [23], [28], for finding vulnerabilities that are due to algorithmic complexity in software applications. Given a program unit that takes inputs of a specified size, the technique computes the input constraints together with actual test input values (of input size) that trigger the worst case complexity of the program. The analysis is parameterized by a *cost model* which gives the cost, such as execution time or memory consumed, for the execution of each instruction, native call, etc. in the program.

Our analysis employs *path guidance policies* to efficiently search for worst-case behaviors. The policies are learned from the worst-case paths obtained with exhaustive symbolic execution of the program at small input sizes. The learned policies are then applied to guide the exploration of the program at larger input sizes. The intuition is that the worst-case paths obtained at small input sizes often follow the same decisions or sets of decisions when executing the conditional statements in the code. Further, the same decisions are "likely" followed by the worst-case paths at larger input sizes. Path policies encode succinctly in a *trie* data structure the decisions taken by symbolic execution along worst-case paths.

A distinguishing feature of our work is that we employ path policies that take into account the *history* of choices made along the path to decide which branch to execute for the conditional statements in the program. For increased precision, the history computation is *context-preserving*, meaning that the decision for each conditional statement depends on the history computed with respect to the enclosing method.

We show experimentally that the proposed history-based policies are often precise enough to decide a unique choice at all decisions. Symbolic execution, guided by these policies, effectively reduces to exploring a single path regardless of input size and it scales far beyond the capabilities of non-guided, traditional symbolic execution for which the number of paths grows exponentially.

To get insight into the worst-case program behavior for *any* input size, we also use techniques to *predict* the worst-case behavior based on the data obtained with the guided analysis. Specifically an off-the-shelf library is used to *fit* a function to the data to obtain an estimate of the complexity as a function of the input size. The results of the function evaluation on increased input sizes are plotted to give insights to the developers into the worst-case complexity of the program. Comparing the estimates to the programmers' expectations or to theoretical asymptotic bounds can reveal vulnerabilities or confirm that a program's performance scales as expected.

We have implemented the technique for the complexity analysis of Java programs, using the Symbolic PathFinder (SPF) tool [28]. However, our approach is general and can be implemented in the context of other languages and/or symbolic execution tools.

Our work is part of a larger effort—the ISSTAC project [4]—which aims to provide an integrated symbolic execution approach for the worst-case complexity and side-channel analysis of Java programs. In this paper, we describe in detail the worst-case analysis component. The technique presented here was able to identify 87.5% of the intended algorithmic complexity vulnerabilities in a recent engagement for the DARPA STAC [1] program. Our specific contributions are: (a) detailed description of history-based guided symbolic execution for worst case analysis, (b) theoretical analysis of proposed approach, (c) evaluation that includes comparison with previous technique and (d) new case studies showing vulnerabilities in challenging, realistic applications.

II. Background and Motivation

A. Symbolic Execution

Symbolic execution is a program analysis technique which executes programs on symbolic instead of concrete inputs [23]. The behavior of a program is described by a *symbolic execution tree* where tree nodes are program states and tree edges are program transitions representing the execution of program instructions. A state includes the (symbolic) values of program variables, a *path condition PC* and a program counter (next instruction). The path condition is a conjunction of constraints that characterizes exactly those inputs that follow the path to the current state. Path conditions are checked using constraint solvers to detect infeasible paths and to generate test inputs for feasible paths. To deal with loops and recursion, typically a bound is put on exploration depth.

B. Motivating Example

Consider the example in Listing 1 taken from an interactive application that extracts commands from `stdin` and allows interaction with a backing store mapping commands to actions.

An adversary is able to exploit the structure of the backing store by constructing inputs in such a way that all commands are stored in the same bucket (i.e. by yielding hash collisions). The vulnerability in the application is that hash collisions are organized in a list, thus, by carefully crafting the inputs, the worst-case behavior can be exercised by an adversary.

```
class Entry {
  String key; Action val; Entry next;
  public Entry(String key, Action val, Entry next) {
    this.key = key; this.val = val; this.next = next;
  }
}
Entry findEntry(String o, int n, boolean insert) {
  for(Entry e = table[n]; e != null; e = e.next) {
    if(e.key.equals(o)) {
      return e;
    }
  }
  if(insert) {
    return list = new Entry(o,null,list);
  }
  return null;
}
class String {
  char[] val;
  // ...
  public boolean equals(Object oObj) {
    // ...
    String o = (String) oObj;
    if(val.length == o.val.length) {
      for(int i = 0; i < val.length; i++) {
        if(val[i] != o.val[i])
          return false;
      }
      return true;
    }
    return false;
  }
}
```

Listing 1: Excerpt of the backing store of an application that processes strings (keys) and associates them with actions.

To find an item in the linked list, the `findEntry` method iterates over it and compares the key strings to its *o* parameter (see Listing 1). Its worst-case behavior is not only determined by the structure of the hashtable—i.e. reduction to a list with worst-case linear look-up time—but the values of the keys play an important role as well: Keys are compared with the `equals` method, that, in the case of strings, performs a character-wise comparison. Consequently, the worst-case behavior of the application happens when a non-existent key is looked up *and* that—when comparing all the entries—the characters constituting the key strings are equal *except* for the last character. This example is typical of applications that use as keys fixed-sized strings (or codes). If the strings are of length k, then the worst-case execution time for `findEntry` is $n \times k$.

Symbolic execution applied to `findEntry` for a small input size can reveal the worst case behavior: The worst-case path follows the edges marked in bold on the CFG of method `equals` illustrated in Figure 1. Note that for condition $c : val[i] \neq o.val[i]$ in the graph (marked with grey) both *true* and *false* branches are taken along the worst-case path.

We also used the exhaustive exploration for small input configurations to automatically derive a *guidance policy* based on the computed worst-case paths. This policy dictates which branch to take during symbolic execution to exercise the worst-case path *for arbitrary input sizes*. Note that, for input size n and string length k non-guided analysis needs to explore k^n paths.

We aim to extract a policy that dictates what branch to take during symbolic execution at any input size. However, a branch policy that simply dictates which choices to *always* make (as in [5]) when executing condition c would be insufficient for our example: the policy would prescribe *both* branches so no savings would be achieved. Thus, in this case, using a simple memoryless policy would be equivalent to exhaustive analysis: memoryless guided symbolic execution (for $k = 15$) becomes quickly intractable for input sizes $n > 3$.

To address this, we employ a more sophisticated guidance policy that takes into account the *history* of decisions taken along the worst-case path. For, e.g., key size $k = 15$, the

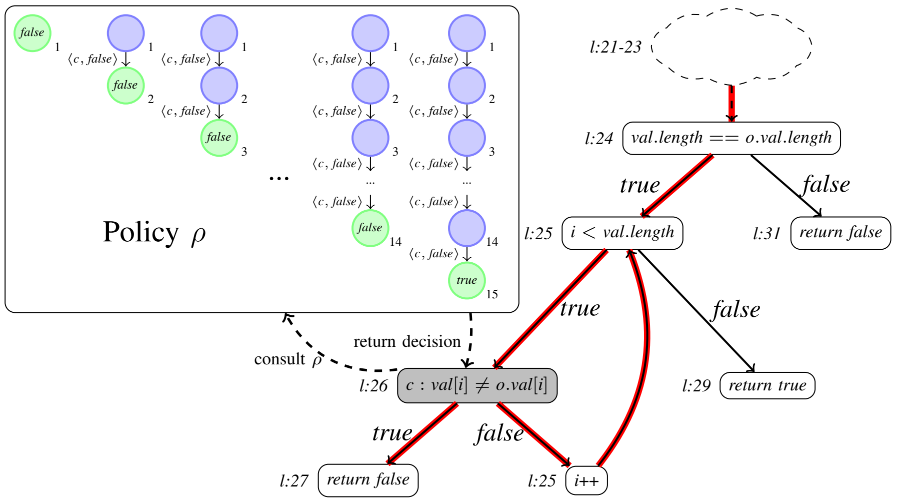

Fig. 1: Partial CFG for the `equals()` method in class `java.util.String`. The worst-case path is highlighted. The policy ρ guides the choices for the decision node marked with gray (for k=15). ρ can be stored as a single path in a trie.

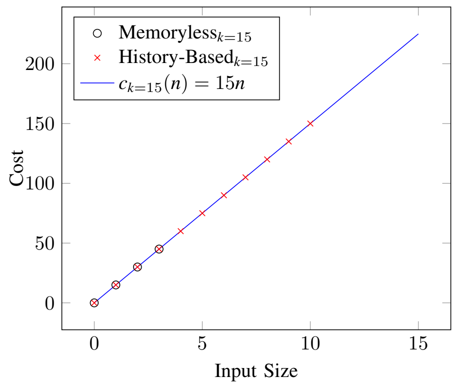

Fig. 2: Data set and prediction model obtained with guided symbolic execution for $k = 15$.

false branch must be taken at the decision on line 30 when comparing the characters for index $0, 1, \ldots, 13$ of the string; the *true* branch must be selected for index 14. Note that this history records the choices along a worst-case path during only one call of method `equals`, but the same pattern is followed for each invocation of method `equals` coming from `findEntry`. Our technique therefore restricts the computation and storage of histories only with respect to a *calling context*. This has two advantages: first, it increases the precision of the analysis, since it can compute different histories for the same condition under different contexts; second, the histories are small since they are local to a particular context.

Our technique computes automatically the *history-based* policy based on an exhaustive exploration at small input sizes ($n = 2$ in this case, but in general over a range as we will describe later). The histories used to define the policy are encoded efficiently in a trie data structure that offers good compression: in this example, all 15 policies can be stored as a single path in a trie, because they are all prefixes of the longest decision history (see Figure 1).

The computed policy is then used to guide the exploration at larger input sizes according to the pattern yielding the worst-case behavior. For the example, when condition c is evaluated during the guided symbolic execution, the symbolic path leading to it is examined to see if the previous up to 13 choices on c (the only symbolic condition in the current context) were all *false*, in which case the policy dictates that the *false* branch should be taken for the current condition (corresponding to the *else* branch in the code). If on the other hand all the previous 14 choices were *false*, the policy dictates to take the *true* branch instead (corresponding to the *then* branch in the code).

The guided search reduces to exploring a single path for an arbitrary number of elements added to the linked list data structure. We can thus expose the worst-case behavior and the concrete inputs that will exercise it. Moreover, the technique computes a model (function) using regression analysis that characterizes the worst-case behavior as shown in Figure 2. As a cost model, we used the number of decisions along an execution path. Note that the history-based policy enables analysis at much larger inputs compared to memoryless guidance (or exhaustive analysis). Further, it enables obtaining concrete inputs that trigger the worst-case behavior at large input sizes, and building a more precise prediction model due to the larger data set (see Figure 2).

III. Complexity Analysis

Our approach relies on symbolically executing the program under analysis for increasing input sizes n and *observing* the terminating path with the largest *cost* for different values of n. We use $P(n)$ to denote the program under analysis parameterized by an integer representing the input size. For instance, for a sorting algorithm, n is the length of the input list to be sorted; for binary search, it is the number of elements added to the tree before searching for a key.

We compute the worst-case path in terms of a *cost model* $\mathcal{C}$, that assigns a cost (e.g. time or memory consumed) to each instruction that is executed, native call, disk access etc. For example, for timing analysis, the worst-case cost is computed by accumulating the instruction costs along the path.

Let c denote a conditional in the control flow graph (CFG) of program $P(n)$. A path ϕ is a finite list of nodes $\pi_1, \ldots, \pi_k$ in the symbolic execution tree. Each node π is a 3-tuple of the form $\pi = \langle c, b, \alpha\rangle$, where c is a conditional in the CFG, b is either *true* or *false* (representing which branch was taken: "then" or "else") and α is the context of the decision; it is represented by the stack frame of the call site. The path can also be empty, represented by ϵ. Let *length*(ϕ) denote the length of the path. We use $\mathcal{C}(\phi)$ to represent the cost of path ϕ.

$\Phi_{P(n)}$ denotes all the paths explored by symbolically executing program $P(n)$.

The 3-tuple $w = \langle \phi, \mathcal{C}(\phi), PC \rangle$ denotes worst case path information for path ϕ including its associated cost $\mathcal{C}(\phi)$ and resulting path condition PC. Note that multiple paths can yield the same (maximal) cost—let $W = \{\langle \phi^*, \mathcal{C}(\phi^*), PC \rangle \mid \forall \phi \in \Phi : \mathcal{C}(\phi) < \mathcal{C}(\phi^*)\}$ denote this set.

A. *Guided Symbolic Execution*

We use guidance policies for the *efficient* exploration of the symbolic execution tree in contrast to an exhaustive analysis, that, although guaranteed to find the worst-case path, does not scale due to the exponential number of paths that need to be explored and the cost of constraint solving. Intuitively, the guidance policies encode the decisions in the control flow graph that the symbolic execution needs to follow to find "likely" complexity vulnerabilities. A set of policies is first generated from an exhaustive symbolic execution for a range of small input sizes, where exhaustive exploration is still possible and the computation of the worst-case paths is precise. For each input size in the range, we obtain a policy that encodes the decisions taken along the respective worst-case path, as mapped on the CFG of the program. The policies are *unified* (as described below) to obtain a more general policy that is then used to *guide* the symbolic execution at larger input sizes. Thus the approach is *guided* towards the paths that maximize the cost in terms of the cost model. All other choices not part of the policy are *pruned* from the search space, enabling a scalable exploration in many cases. This is explained in more detail below.

A policy ρ is a function mapping a CFG branch c to a choice $b \in \{\bot, \{true\}, \{false\}, \top\}$. Initially, ρ maps all conditions of the program to $\bot = \{\}$ (the empty set). We use $\top = \{true, false\}$ to denote the special value when the policy is inconclusive as to which choice to make at branch c.

We also introduce *policy unification*: If ρ_1 and ρ_2 are two policies for program P, then their unification, $\rho_\cup = \rho_a \cup \rho_b$, is defined as the union of their individual mappings of conditions to decisions. E.g., for condition c, if $\rho_a(c) \rightarrow \{true\}$ and $\rho_b(c) \rightarrow \{false\}$, then $\rho_\cup(c) \rightarrow \{true, false\}$.

We say that a policy is *deterministic* if it contains no $\top$- or $\bot$-values. In this case, the guided symbolic execution results in only one program path being explored. Note that even if a policy is not deterministic, it is still very useful, because it can prune large portions of the exploration space (for the conditions that are decided uniquely).

The overall algorithm for guided worst-case complexity analysis is shown in Procedure 1—it has two phases: (i) Policy generation; and (ii) Policy-guided search.

The algorithm takes as input a program P, two small bounds on input size $L \leq H$ to be used for policy generation, and a large input-size bound N to be used for guided symbolic execution using the generated policies. Furthermore, $\mathcal{C}$ specifies the cost model and κ is used for computing a rank for each policy as described in detail later. For simplicity of presentation, we assume that we use the same cost $\mathcal{C}$ in both phases of

Procedure 1 Worst-case complexity analysis.

```
Input: Program $P(n)$, input size bound $N$, policy gen. input
    sizes $L$, $H$ s.t. $L \leq H < N$, cost model $\mathcal{C}$, policy score $\kappa$
    {Phase (i)}
1:  $\rho_\cup$ initialized to $\bot$ for all CFG conditions $c$
2:  for $j \leftarrow L$ to $H$ do
3:      $W_e \leftarrow exhaustiveWCA(P(j), \mathcal{C})$
4:      $\rho_{best} \leftarrow null$
5:      for all $\langle \phi, \mathcal{C}(\phi), PC \rangle \in W_e$ do
6:          $\rho \leftarrow computePolicy(\phi)$
7:          if $\rho_{best}$ is $null$ then
8:              $\rho_{best} \leftarrow \rho$
9:          else if $\kappa(\rho) > \kappa(\rho_{best})$ then
10:             $\rho_{best} \leftarrow \rho$
11:     $\rho_\cup \leftarrow \rho_\cup \bigcup \rho_{best}$
    {Phase (ii)}
12: $D \leftarrow \varnothing$
13: for $i \leftarrow 1$ to $N$ do
14:     $W_g \leftarrow guidedSymExe(P(i), \rho_\cup, \mathcal{C})$
15:     $cost_i \leftarrow \mathcal{C}(\phi)$ s.t. $\langle \phi, \mathcal{C}(\phi), PC \rangle \in W_g$
16:     $D \leftarrow D \cup \langle i, cost_i \rangle$
17: $\langle f, r^2 \rangle \leftarrow regressionAnalysis(D)$
18: Output $\langle f, r^2 \rangle$, input constraints and solutions.
19: return
```

the algorithm. However, in general this is not necessary. For example, for policy generation, we may want to compute the paths that visit the largest number of symbolic conditions, or the largest number of loops, taking into account also the nested loops. Although these paths may not be "costly" in terms of execution time at small input sizes, they may be the "most promising" in indicating the worst-case behavior for larger input sizes. It can thus make sense to generate the guidance policies using these paths while in the second phase to perform iterative guided symbolic execution using the cost model that, e.g., accumulates the execution time for each instruction. Our implementation supports these options to allow for easy experimentation. For our experiments, we assumed that the cost is the number of decisions along a path.

In Phase (i), policies are computed from the worst-case path(s) obtained from input sizes $L..H$. Note that, for a specific input size there can be multiple paths that yield the same worst-case cost. For each of these, we compute their *rank*, a measure that quantifies how promising the policy is based on a ranking model κ. E.g., κ can rank the policy based on how many decisions it can uniquely resolve, i.e. a policy is better the less $\top$- and $\bot$-values it contains.

Policy ρ_{best} stores the best policy for input size j according to κ. Policy $\rho_\cup$ stores the successively *unified* policy obtained from exhaustive analysis over input sizes $L..j$. When all worst-case paths have been evaluated for a given input size, j, ρ_{best} is unified with $\rho_\cup$. When Phase (i) terminates, $\rho_\cup$ stores the unified policy for input sizes $L..H$.

In Phase (ii), the unified policy $\rho_\cup$ is used for iteratively computing data points for the regression analysis, for increasing input sizes $1..N$. The data points are obtained by a guided symbolic execution, that upon every decision encountered in the symbolic execution, consults the policy to resolve it, i.e. it selects the choice stored in the policy to be explored next, disregarding all other choices at that decision.

Procedure 2 computePolicy

```
Input: Worst-case path $\phi$
Output: Policy $\rho$
1: $\rho$ initialized to $\perp$ for all CFG conditions $c$
2: for all $\pi_k = \langle c, b, \alpha \rangle$ where $k = 1, ..., length(\phi)$ do
3:     $\rho(c) \leftarrow \rho(c) \cup b$
4: return $\rho$
```

a) Computing Policies: Given a worst-case path, procedure `computePolicy` (Procedure 2) iterates through the list of decisions along the path and updates the policy with the decision made at each CFG branch along the path.

Procedure 3 guidedSymExe

```
Input: Program $P$, policy $\rho$, cost model $\mathcal{C}$
Output: $W = \{\langle \phi, \mathcal{C}(\phi) \rangle_1, ...\}$
1: Run symbolic execution on $P$ and record worst-case paths
   in set $W$
2: for all $\pi = \langle c, b, \alpha \rangle$ about to be explored do
3:     $choice \leftarrow \rho(c)$
4:     if $choice \neq \perp$ and $choice \neq \top$ then
5:         Explore $b = choice$ for $c$ in $\pi$
6:     else
7:         Explore both $b = true$ and $b = false$ for $c$ in $\pi$
8: return $W$
```

b) Policy Guided Exploration: Procedure 3, `guidedSymExe`, guides the search using policy ρ. Whenever a conditional instruction, c, is about to be evaluated, $\rho(c)$ is used to resolve the choice. If a unique resolution exists, only that one is explored while all other choices are pruned from the search. If $\perp$ or $\top$ is prescribed as next choice both branches need to be considered and the guided search degenerates to exhaustive exploration at that condition.

c) Ranking Policies: The initial exhaustive explorations each potentially yields a set of worst-case paths with the same maximal cost. From these, policies are computed and ranked in terms of the policy's ranking model, κ (Procedure 4). To compute the policy rank, all branch instructions of the program are used to determine whether it is deterministic. For each deterministic choice, the policy rank is incremented. Essentially, the rank quantifies how many deterministic choices can be made with the policy.

Ranking the policies is needed, because they differ in how well they resolve choices in the guided search. For example, for Insertion Sort, multiple paths yield the same worst-case cost for same input size, but only one can be used to compute a deterministic policy. Our experiments have shown that this ranking is effective allowing us to compute a deterministic policy in most cases for the evaluated examples. We plan to experiment with more complex scoring models that take into account other statistics, e.g., the number of loops visited, the maximum depth of visited nested loops and other metrics.

Procedure 4 κ

```
Input: Policy $\rho$
Output: Policy rank
1: $rank \leftarrow 0$
2: for all $c$ in $P(n)$ do
3:     $res \leftarrow \rho(c)$
4:     if $res \neq \perp$ and $res \neq \top$ then
5:         $rank \leftarrow rank + 1$
6: return $rank$
```

d) Regression Analysis: Finally, the data points are used for regression analysis (calling procedure `regressionAnalysis`) to yield the characterization of the worst-case complexity. In practice, we rely on multiple linear regression using the Ordinary Least Squares (OLS) method: it computes a function that minimizes the differences of the original data set and the predicted data points from the function. As is customary with this technique, functions are evaluated based on the *goodness-of-fit*, which, conventionally is the r^2 value, i.e. the coefficient of determination defined as $1 - (S/T)$, with S representing the sum of squared residuals and T the total sum of squares. Thus, the closer r^2 is to 1, the better is the fit.

The found complexity classes together with the input constraints and solutions for each n are output. The user has an important role in our approach as he/she needs to examine the data to determine vulnerabilities. To facilitate this process, our approach provides the visualization of the generated data using multiple views that allow the analysis to zoom in and out on specific portions of the plotted graphs. However, the algorithm may be stopped early (before all the policies are used) when a vulnerability is confirmed or when the analyst believes that there is no indication of vulnerabilities.

B. History-based Policies

The policy described above is "memoryless" in the sense that it does not keep track of previous choices taken along a path. As demonstrated in the example in Section II-B, memoryless policies cannot always be used for resolving decisions deterministically. As another example consider the case for Merge Sort where the worst-case behavior manifests itself if alternating choices are made at a certain condition. To capture this pattern, the policy needs to memorize the previous decisions. In fact, the worst-case behavior of algorithms is often not determined by selecting a unique choice at a condition.

A distinguishing factor of our work is that we use policies that take into account *decision histories* to address the limitations of memoryless policies. Intuitively, a decision history for a node π encodes the sequence of decisions for each conditional instruction taken along a worst-case path leading to π. To improve precision and also keep the size of the generated policy manageable, we use *context-preserving decision histories*, denoted $\mathcal{H}(\pi)$ encoding the sequence of decisions taken in the context α of π, i.e. $\mathcal{H}(\pi) = \pi_1, \pi_2, .., \pi_n$ such that $\forall\, \pi_i \in \mathcal{H}(\pi) : \alpha_i = \alpha$ and $\pi_1, \pi_2, .., \pi_n, \pi$ is the suffix of a path leading to π in the symbolic execution tree.

We define $\downarrow(\mathcal{H}(\pi))$ as the sequence of decisions in $\mathcal{H}(\pi)$ from which we discard the α context component. For example, for $\mathcal{H}(\pi) = \langle c_1, b_1, \alpha\rangle$, $\langle c_2, b_2, \alpha\rangle$ .. $\langle c_5, b_5, \alpha\rangle$, $\downarrow(\mathcal{H}(\pi))$ is the sequence $\langle c_1, b_1\rangle$, $\langle c_2, b_2\rangle$ .. $\langle c_5, b_5\rangle$. Thus $\downarrow(\mathcal{H}(\pi))$ can be seen as an *abstraction* of $\mathcal{H}(\pi)$.

Furthermore, we use a history size h to define $\mathcal{H}_h(\pi)$ as the sequence formed by the last h elements of $\mathcal{H}(\pi)$. Intuitively, the history size h quantifies how much from the history we want to take into account when computing guidance policies. For the example above and $h = 3$, $\downarrow(\mathcal{H}_3(\pi)) = \langle c_3, b_3\rangle, \langle c_4, b_4\rangle, \langle c_5, b_5\rangle$.

We can now formalize the concept of a *history-based policy*. A history-based policy $\rho_{\mathcal{H}}$ extends the memoryless policy ρ with decision history parameter $\downarrow(\mathcal{H}_h(\pi))$ of size h. It is a function mapping a CFG branch c and $\downarrow(\mathcal{H}_h(\pi))$ to a choice $b \in \{\perp, \{true\}, \{false\}, \top\}$. Thus, although we compute the decision histories with respect to context α, we only keep abstract (suffix) histories in the policy.

The intuition for using a context-preserving history, is that the policy will be kept small and only contain observed behaviors of the enclosing method of c. The intuition for using $\downarrow(\mathcal{H}_h(\pi))$ is that when applying the policy in Phase (ii), we match on behaviors of size h confined to the method disregarding the calling context making it more general. We plan to experiment in future work with policies that keep the context un-abstracted. However such policies will be much larger and less general.

The notion of a unified policy—as introduced for memoryless policies—naturally also extends to history-based policies.

In practice, we observe that many histories that define the policy for a condition c share the same prefix, and thus they are stored *efficiently* as a trie as mentioned in Section II-B. An interesting direction for future work is to generalize the tree into an automaton via tree folding [17].

Note that the history size can be defined globally to be the same (usually small) fixed value for all the decisions. The analyst can then try iteratively to find vulnerabilities using increasing values of h. If $h = 0$, this is equivalent to a memoryless policy. If a global value for history size is not specified, then ρ is defined with respect to $\downarrow(\mathcal{H}(\pi))$, i.e. the histories are not truncated (they are still bounded since the symbolic execution is bounded).

To accommodate history-based policies in our technique, we use the same algorithms as the ones presented in the previous sections but updated for: (1) policy updating and (2) policy guided search.

For (1), we update line 3 in Procedure 2 as follows:

$$\rho(c, \downarrow(\mathcal{H}_h(\pi_k))) \leftarrow \rho(c, \downarrow(\mathcal{H}_h(\pi_k))) \cup b$$

The approach is the same as in the memoryless policy computation except that, for each decision, π_k, its associated decision history $\downarrow(\mathcal{H}_h(\pi_k))$ is also accounted for. Note that for multiple calls to the same method, the behaviors observed are simply added to the policy which summarizes all the behaviors observed for the condition c.

For (2), we update line 3 in Procedure 3 as follows

$$choice \leftarrow \rho(c, \downarrow(\mathcal{H}_h(\pi)))$$

We also require that Procedure 3 takes a history-based policy $\rho_{\mathcal{H}}$ as argument. Note that during the guided symbolic execution it may be the case that the history $\mathcal{H}(\pi)$ is much longer than any of the histories examined when the policy was built (since the guided symbolic execution is performed at a larger input size). However, we are able to use the policy computed based on smaller histories to guide the execution at larger histories, since we only use the smaller suffix $\downarrow(\mathcal{H}_h(\pi))$ to compute the next decision at c. Also note that if a global value is not specified, h is taken to be the height of the trie corresponding to the policy at the condition c.

We have also experimented with an *adaptive strategy* for computing values of h at each c such that the resulting policy has the best rank (i.e. leads to the most deterministic choices). Intuitively, this can be computed iteratively increasing the history size (starting from 0 up to the trie height) until all choices can uniquely be determined or the trie height has been reached. However in our experiments we have not noticed any significant difference with this more sophisticated strategy.

C. Theoretical Guarantee

We provide a theoretical guarantee that when we unify policies from L up to a sufficiently large size, heuristic exploration using that policy is guaranteed to lead to the worst case for any input size (starting with L).

Theorem 1 *Let $\rho_{\cup}^{L..H} = \bigcup_{n=L..H} \rho_n$ denote the unification of the policies obtained from the analysis at input sizes L..H, for same history size h. Then there exists a sufficiently large M such that the policy $\rho_{\cup}^{L..M}$ accurately predicts the worst-case path for any input size that is greater or equal to L.*

Proof: First observe monotonicity of policy generation. We define $\rho_1 \subseteq \rho_2$ as $\forall_{i \geq L}\{\Phi_{i,\rho_1} \subseteq \Phi_{i,\rho_2}\}$, where $\Phi_{i,\rho}$ is the set of paths explored with policy ρ at input size i. Unification of policies leads to increased coverage of program behaviors: if $\rho_{\cup}^{L..n+1} = \rho_{\cup}^{L..n} \bigcup \rho_{n+1}$ then $\rho_{\cup}^{L..n+1} \supseteq \rho_{\cup}^{L..n}$, since ρ_{n+1} can only add more behaviours that are allowed. Since the number of choices for the policy is finite and the history size is fixed, there is a finite number of possible policies. Hence, there exists an M for which $\rho_{\cup}^{L..M}$ is 'largest' according to $\subseteq$ and thus includes the worst-case path for any input size that greater or equal to L. ■

Note that in practice we observed that often we could find a deterministic policy for some $L = H$ that would correctly explore the worst-case path up to N in our algorithm. This is primarily attributed to the fact that policies are often *invariants*, i.e. the behaviors they prescribe are unaffected by input size as is the case for our motivational example, Find Entry. For this example, the policy obtained at size 2 is the same as the one obtained at size 3 etc. but different than the policy obtained at sizes 0 or 1. Thus the unified policy (over 2..3) is still deterministic. The same pattern was observed in the other examples (but not in Merge Sort—see discussion in the next section). We therefore fixed $L = H$ in our experiments. This simplification reduced significantly the parameter space for our analysis while still computing the correct complexity classes for all examples.

IV. Evaluation

We have implemented the presented analysis in Symbolic Pathfinder (SPF) [28], a symbolic execution tool for Java bytecode programs that is part of the Java Pathfinder (JPF) toolset [21]. The implementation uses two JPF listeners corresponding to the two phases described in Procedure 1. In addition, a JPF shell wraps the analysis and extracts the relevant data for function fitting and visualization (with JFreeChart [20]) of the results. The regression analyses used for function fitting are implemented using the Ordinary Least Squares multiple linear regression component of Apache Commons Math [8]. The trie data structure representing the histories defining the policies can be stored on disk and reused.

We evaluate the presented analysis against a set of challenging Java benchmarks provided to us by Raytheon BBN and CyberPoint LLC as part of an engagement for the DARPA STAC [1] program. In total, we identified 21 complexity vulnerabilities (87.5% of intended vulnerabilities) in these programs with this technique presented in this paper. We describe a sub-set here (they are available on request).

URI Verifier is a component in a web blogging platform, Blogger; ZIP Decompressor is a component in a text analysis and encryption program, TextCrunchr; Find Entry is the motivating example; NGram Score is another component from TextCrunchr that scores all n-grams in substrings of length n against a database of n-grams; Password Checker is a component that checks a 32 byte hash of a password against the hashes of the last N passwords to prevent re-use; Database B-Tree is a component in the LawDB system, which uses a B-tree for storing an index over law personnel.

We also analyzed the benchmarks from WISE [5] representing the standard implementations of classic algorithms.

We devised a workflow for applying our technique in practice. The first step is to find a "good" policy by iteratively trying the analysis for small input ranges (s.t. $H = L$) and $h = 0$ and then studying them in terms of the ranking score, κ, and visually using a projection on a rendered CFG. If a policy has few deterministic choices (low κ), we increase the history size, h, to see if it leads to a more deterministic policy (i.e. increasing κ). We prioritize the policies over values of $L = H$ and h according to κ. When the policies have been evaluated, we apply them—in priority order—with guided analysis up to N and study the fitted functions obtained from the data collection. We found that, in practice, applying this workflow is effective and can be largely automated by a script.

The results of our experiments (for the best parameters we found) are shown in Table I. The techniques that we compared are exhaustive symbolic execution i.e. without policies (row *Exh.*), memoryless guided analysis (*m.l.*) and, where applicable, history-based guided analysis. The experiments were run on a Dell XPS 13 notebook with a 6th Generation Intel Core i7 processor and 8GB RAM running Ubuntu Linux. We used Z3 [10] as solver for all experiments.

The results show that, as expected, exhaustive symbolic execution is often only feasible up to small input sizes but guided symbolic execution scales much better allowing us to find the correct worst-case behavior for these examples. For the NP-complete Traveling Salesman Problem, we can explore up to $N = 7$, but note that regardless of input size, our technique reduces to exploring a *single* path since we can compute a deterministic policy that correctly resolves all choices. When memoryless policies are incapable of resolving all choices, the results show that the history-based policies are beneficial, enabling more deterministic resolutions of decisions.

Another observation is that different policies obtained at *different* input sizes can still lead to the discovery of the correct complexity classes, although they are not invariant. This is illustrated by Merge Sort: For input size 7, the policy prescribes one alteration scheme when merging the two sub partitions, while for input size 8, the opposite alteration scheme is prescribed by the policy. However, when unifying the two policies, the result is a policy that is not deterministic, that while correct, does not prune many choices because it prescribes both contradictory alteration schemes. For input size 7 (8), exploration at input size 30 reduces to a single path with 128 (132) choices resolved (hence none unresolved choices). For the policy obtained as the unification of input size 7 and 8, exploration at input size 30 only resolves 31 choices, and leaves 1844 choices unresolved (totalling 923 paths explored). This also suggests a direction for future work, where we can use different unification scheme for policies, e.g. we can simply keep the union of policies discovered at smaller sizes and apply them at larger sizes. Such scheme would still ensure that we eventually find the worst-case path at any size.

Database B-Tree used a nondeterministic policy. However the guided analysis scales by several orders of magnitude compared to the exhaustive analysis.

In order to fit a function at least 5 data points are needed. For complex algorithms, such as the Traveling Salesman Problem, this would be impossible without policy-guided exploration. In other cases, obtaining more data points increases precision of the fitting function. From a collection of fitting functions corresponding to the standard complexity classes, column "Complexity" shows the best fit found based on a combination of highest r^2 and visual interpretation of the results.

Note that it is not always possible to determine the correct complexity class based on r^2, even though the correct worst-case path was discovered. For example, for Red Black Tree, a power function is a slightly better fit than a logarithmic model (see Figure 3). For such cases the analyst needs to visualize the data and select the best fit (clearly logarithmic here). In the

TABLE I: Experimental results. For each benchmark we list nr. of explored paths and run-time for a series of input sizes. Policies were inferred at input size N_{pol}. The h column specifies the history size or *m.l.* when a memoryless policy sufficed.

Benchm.	**Set-up**			**Input Size (N)**												**Complexity**	r^2
	$L{=}H$	h		**1**	**2**	**3**	**4**	**5**	**10**	**15**	**20**	**30**	**100**	**250**	**1000**		
`Blogger URI Verifier`	**Exh.**		**Paths**	55	2213	114533	-	-	-	-	-	-	-	-	-		
			Time	0:02	0:25	26:46	-	-	-	-	-	-	-	-	-		
	1	*m.l.*	**Paths**	8	57	155	351	743	-	-	-	-	-	-	-	$\mathcal{O}(n^2)$	0.99986
			Time	0:00	0:02	0:31	4:45	45:09	-	-	-	-	-	-	-		
`TextCrunchr ZIP Decompressor`	**Exh.**		**Paths**	3	4	5	6	7	12	17	22	32	102	252	1002		
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:01	0:06	1:27		
	1	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	1	1	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:01	0:06	1:28		
`Find Entry` ($k = 15$)	**Exh.**		**Paths**	16	376	11656	-	-	-	-	-	-	-	-	-		
			Time	0:01	0:07	4:59	-	-	-	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	16	376	11656	-	-	-	-	-	-	-	-	-	(too few predictors)	
			Time	0:01	0:07	4:09	-	-	-	-	-	-	-	-	-		
	2	14	**Paths**	1	1	1	1	1	1	1	1	1	1	1	-	$\mathcal{O}(n)$	1.0000
			Time	0:01	0:01	0:02	0:02	0:02	0:04	0:08	0:12	0:24	6:00	2:00:32	-		
`TextCrunchr NGram Score` (trigrams)	**Exh.**		**Paths**	4	13	40	121	364	88573	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:01	0:02	3:15	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	4	13	40	121	364	88573	-	-	-	-	-	-	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:00	0:00	0:01	0:02	5:10	-	-	-	-	-	-		
	2	2	**Paths**	1	1	1	1	1	1	1	1	1	1	1	1	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:02	0:15	7:07		
`Password Checker` ($k = 32$)	**Exh.**		**Paths**	33	1057	33825	-	-	-	-	-	-	-	-	-		
			Time	0:01	0:04	2:00	-	-	-	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	33	1057	33825	-	-	-	-	-	-	-	-	-	(too few predictors)	
			Time	0:00	0:03	2:04	-	-	-	-	-	-	-	-	-		
	2	31	**Paths**	1	1	1	1	1	1	1	1	1	1	-	-	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:04	-	-		
`LawDB Database B-Tree`	**Exh.**		**Paths**	3	13	75	541	4683	-	-	-	-	-	-	-		
			Time	0:01	0:01	0:01	0:01	0:07	-	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	2	3	4	3	3	4	5	5	6	8	583	-	$\mathcal{O}(\log n)$	0.99755
			Time	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:29	06:23	-		
`Sorted Linked-List` insert	**Exh.**		**Paths**	1	2	6	24	120	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:01	0:01	-	-	-	-	-	-	-		
	3[†]	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	-	-	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:01	0:01	0:01	0:01	0:01	0:02	0:04	0:10	58:51	-	-		
`Heap` insert (JDK 1.5)	**Exh.**		**Paths**	1	2	4	12	36	20736	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:00	0:01	0:46	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	1	1	$\mathcal{O}(\log n)$	0.99699
			Time	0:00	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:02	0:09	0:53		
`Red-Black Tree` search	**Exh.**		**Paths**	3	10	42	216	1320	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:01	0:03	-	-	-	-	-	-	-		
	8	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	1	-	$\mathcal{O}(\log n)$	0.99837
			Time	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:02	0:25	7:09	-		
`Quicksort` (JDK 1.5)	**Exh.**		**Paths**	1	2	6	24	120	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:00	0:01	-	-	-	-	-	-	-		
	8	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	-	-	$\mathcal{O}(n^2)$	0.99997
			Time	0:00	0:01	0:01	0:01	0:01	0:01	0:02	0:06	0:09	37:42	-	-		
`Binary Search Tree` search	**Exh.**		**Paths**	1	3	13	75	541	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:01	0:02	-	-	-	-	-	-	-		
	3	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	-	-	-	$\mathcal{O}(n)$	1.0000
			Time	0:00	0:01	0:01	0:01	0:01	0:01	0:03	0:05	0:13	-	-	-		
`Merge Sort` (JDK 1.5)	**Exh.**		**Paths**	1	2	6	24	120	3628800	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:00	0:01	2:06:22	-	-	-	-	-	-		
	7	*m.l.*	**Paths**	1	1	1	1	1	251	-	-	-	-	-	-	$\mathcal{O}(n \log n)$	0.99591
			Time	0:00	0:00	0:00	0:00	0:00	0:02	-	-	-	-	-	-		
	7	1	**Paths**	1	1	1	1	1	1	1	1	1	1	1	-	$\mathcal{O}(n \log n)$	0.99941
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:01	0:11	2:03	-		
	8	1	**Paths**	1	1	1	1	1	1	1	1	1	1	1	-	$\mathcal{O}(n \log n)$	0.99962
			Time	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:00	0:11	1:33	-		
`Bellman-Ford`[‡]	**Exh.**		**Paths**	1	2	63	-	-	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:02	-	-	-	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	1	1	1	1	1	1	-	-	-	-	-	-	$\mathcal{O}(n^3)$	1.0000
			Time	0:00	0:00	0:00	0:01	0:02	6:19	-	-	-	-	-	-		
`Dijkstra's`[‡]	**Exh.**		**Paths**	1	1	4	56	2592	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:00	0:10	-	-	-	-	-	-	-		
	3	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	-	-	-	$\mathcal{O}(n^2)$	1.0000
			Time	0:00	0:00	0:00	0:00	0:00	0:01	0:03	0:11	1:16	-	-	-		
`Traveling Salesman`[‡]	**Exh.**		**Paths**	1	1	3	297	-	-	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:02	-	-	-	-	-	-	-	-		
	3	*m.l.*	**Paths**	1	1	1	1	1	-	-	-	-	-	-	-	$\mathcal{O}(n!)$	0.99935
			Time	0:01	0:01	0:01	0:02	0:04	-	-	-	-	-	-	-		
`Insertion Sort`	**Exh.**		**Paths**	1	2	6	24	120	3628800	-	-	-	-	-	-		
			Time	0:00	0:00	0:00	0:00	0:01	1:37:46	-	-	-	-	-	-		
	2	*m.l.*	**Paths**	1	1	1	1	1	1	1	1	1	1	1	1	$\mathcal{O}(n^2)$	1.0000
			Time	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:01	0:02	0:05	1:10		

[†] In [5], the policy is found at input size 2. There appears to be a mismatch between the concept of input size for their policy and for their measurements. As N, we use the size of the list into which an element is inserted. It does not include that inserted element.

[‡] The graph used in these examples are constructed as a grid, thus the number of edges is $|E| = |V|^2 - |V|$. Therefore, the complexity of Bellman-Ford is $|V|^2\,(|V| - 1)$, i.e. cubic as found by the analysis. The Dijkstra's algorithm implementation used does not use a min-priority queue, thus has theoretical complexity $|V|^2$, which is confirmed by our technique.

future, we plan to experiment with other regression analyses that allows to enforce constraints on the parameters, such as the non-negative least squares method, to obtain better fits.

Note also that a related approach, WISE, uses guided symbolic execution for worst-case analysis, although the work is done in the context of dynamic (not static as we do here) symbolic execution and also their heuristics are different, as they use the concept of branch generators, which do not take into account the history of computation within calling contexts. Our results on the benchmarks from WISE show that our memoryless analysis is equivalent to WISE [5] as we obtain similar results for all the examples where a history is not needed. We do improve on WISE for those examples where an extra row using history is shown, e.g. for Merge Sort.

With the policy found without history, it is not feasible to explore the search space for $N > 10$—WISE obtains a similar result. In fact the memoryless policy is not much better than exhaustive analysis for this example. However, when using a history-based policy, the guided search prunes all paths except the worst-case path—it is a deterministic policy that optimally resolves choices enabling exploration up to $N = 250$. For this example, taking into account the calling context is also important: the policy specifies which choices to make for a condition in the `Arrays` class, which is in a loop. Since the Merge Sort implementation is recursive, these decisions must only happen for the current call to Merge Sort, i.e., the context must be taken into account for correct resolution of decisions. We also conjecture that WISE would not work on the other examples that require context-preserving histories.

A. *Vulnerability Analysis*

a) URI Verifier: We found a vulnerability in the URI verification component of BLOGGER, a web-blogging platform.

We applied symbolic execution to the URI Verifier Component using a symbolic URI string (character array with symbolic values). N denotes the length of the URI. We quickly found for $L = H = 1$ and $h = 0$ a deterministic policy that would enable guided analysis to explore up to $N = 5$. The data points were sufficient for the regression analysis (see Figure 4) with the results indicating that the algorithm is exponential with respect to the length of the URI; an unintended algorithmic design flaw in the order of the complexity class. The vulnerability is that an adversary has full control of the URI contents, and therefore can trigger unusually long processing times effectively making the service unresponsive to its benign users (DoS attack).

Our technique also outputs concrete inputs that expose the vulnerability: they show that the URI must be composed of all lower-case and unique characters. We validated it by making the HTTP request `http://localhost:8080/abcde`, which indeed triggers the exponential running time that can be exploited. For URIs not conforming to the constraints found by our technique, the running time is linear and not exploitable.

b) ZIP Decompression: We also found a vulnerability in the ZIP Decompression component of TEXTCRUNCHR, a text encryption and analysis program.

We applied our technique by symbolically constructing a ZIP file, with the number of files controlled by a symbolic variable. Since TEXTCRUNCHR only allows a single input file, the input size N represents a nesting of ZIP files, i.e. a ZIP file containing another ZIP file etc. We obtained a deterministic policy for $L = H = 2$ and $h = 0$ and then performed guided symbolic execution up to input size 400 with increments of 10. The results—shown in Figure 5—indicate that the number of files (or ZIP nesting levels) to be processed is *unbounded*, since the number of files processed is linear in the nesting level ($1..N$). In other words, we can keep adding new elements to the queue during each loop iteration and maintain a queue size that does not exceed the bound.

Solving the constraints show that the (symbolic) content of a ZIP file does not need to contain other elements except for the nested ZIP file—this will yield a queue with at maximum two elements at any given time. TEXTCRUNCHR bounds the input file to 5MB, and we used this information along with the constraints to construct a ZIP quine (ZIP that extracts a copy of itself) of 440 bytes. With this file, the program does not terminate, exposing a complexity vulnerability. The vulnerability here is that the data guard is *incorrect*: instead of bounding the maximum queue size, the bound should have been on the number of files processed.

V. RELATED WORK

The closest to our work is WISE [5], which uses dynamic symbolic execution and "branch generators" for complexity analysis. As discussed for the experiments the WISE approach may fail to compute the worst-case execution behavior of a program, since it may not prune enough executions for larger input sizes. For example, for Merge Sort, WISE's policy still leaves an exponential search for worst-case executions [5] while with our approach we can efficiently find the worst-case behavior. This is due to the fact that our policies are both history-based and context-preserving, which enable us to efficiently determine worst-case bounds for programs that could not be analyzed before. Another, minor, difference is that our approach is implemented in the context of classical symbolic execution. More significantly, we integrate *function fitting* to derive the worst-case estimates. We found that without function fitting, it is very difficult to predict the worst-case behavior at large input sizes, since symbolic execution, even with guidance policies, is difficult to scale for very large inputs. Further, function fitting provides estimates that are very useful to the analyst who can check them visually and further attempt to prove them formally using theorem proving.

Another related work [34] employs symbolic execution for load testing by performing an *iterative* analysis for increasing exploration depth. At each iteration, the paths that are most promising in terms of a resource consumption measure are selected to be explore at the next iteration, while the other paths are discarded. Thus, similar to our method, this work is also *guided*, however it could not be used directly for estimating the worst-case algorithmic behaviour. To see this note that all the paths explored up to the same depth will have

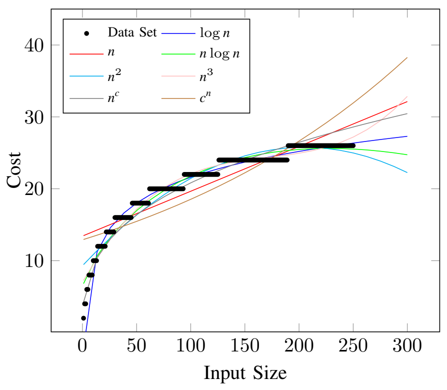

Fig. 3: Fitting functions for Red Black Tree Search.

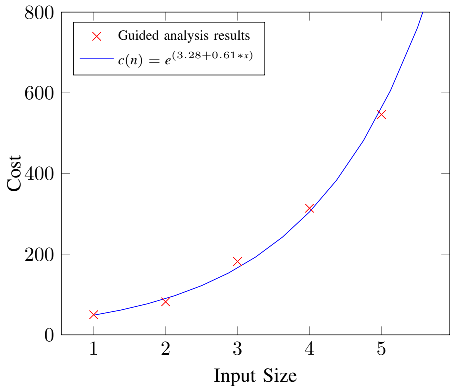

Fig. 4: Results for URI Verifier.

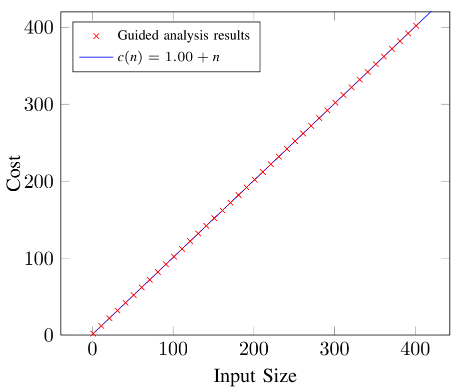

Fig. 5: Results for ZIP Decompression.

the same number of steps, leading to the same algorithmic execution time, thus leading to no pruning.

Probabilistic symbolic execution is used in [7] to infer the performance distribution of a program according to given usage profiles. The technique aims to obtain a diverse set of program behaviors by guiding the execution along high- and low-probability program branches; it further uses loop unrolling to try to force the longest execution through the loops. Thus unlike our work, the technique does not learn how to guide execution at larger configurations from the analysis of smaller configurations. As a result the technique does not yet scale for large programs; this is due also to the more involved probabilistic computation. An interesting direction for future work would be to combine the probabilistic exploration from [7] with the policy learning that we employ to obtain diverse policies for guided execution at larger configurations.

Static analysis has been used to compute conservative bounds on looping programs [2], [15], [16], by computing bounds over ranking functions. These are functions on loop variables which keep decreasing with each loop iteration and are difficult to derive automatically when the functions are non-linear. Static analysis produces an upper bound on the worst-case computational complexity, while our approach produces a lower bound. Furthermore our approach generates inputs that expose the worst-case behavior and can be run independently by the developers. An interesting direction for future work is to check the estimates we obtained with function fitting using a static analysis, possibly guided also by the symbolic constraints generated by symbolic execution.

Related to our work is the large body of research in worst-case execution time (WCET) analysis—in particular for real-time systems [18], [19], [26], [31]. There are many techniques for WCET estimation leveraging model checking [9], [27], [29], symbolic execution [22], [24], integer linear programming [25], [29], abstract interpretation [33], or a combination, e.g., [27]. An overview can be found in [32]. Our approach is orthogonal to this body of work since we target complexity analysis. Further, most of these techniques assume that loops have finite bounds, independent of input size, while we estimate the bounds for programs with input-dependent loops.

Profilers [3], [14], [30] are typically used for performance analysis of programs. Profilers collect various timing statistics by sampling periodically (or continuously) the program states. The approach from [13] computes the observed computational complexity by using profiled data in conjunction with curve-fitting to compute empirical computational complexity, similar to our function fitting procedure. However, contrary to our technique, profilers are inherently limited by the number of tests used during profiling. Moreover, the profiled data used to compute the complexity might not constitute the worst-case complexity. Instead we aim to compute worst-case bounds in a more systematic way, using guided symbolic execution.

VI. Conclusions and Future Work

We presented a symbolic execution technique for analyzing the worst-case complexity of programs. The symbolic exploration is guided by policies that take into account the history of execution along the worst-case paths. We implemented the technique and showed its promise in practice.

In the future, we plan to improve the policy-guided search by adding an additional pruning based on *decision frequency invariants*, which relate the frequencies with which decisions are made at a CFG branch. Such invariants can be inferred automatically using e.g. Daikon [11]. If the policy is inconclusive at a condition, the path invariant can be consulted to determine if one of the choices will violate the invariant (in which case it is pruned from the search). We also plan to extend the analysis to cater for a more fine-grained, instruction- and context-dependent modeling of execution costs.

Acknowledgment

We thank Raytheon BBN and CyberPoint LLC for providing the use cases. This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] DARPA's Space-Time Analysis for Cybersecurity program. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[3] G. Ammons, J. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *ECOOP 2004 - Object-Oriented Programming*, pages 170–194, 2004.

[4] D. Balasubramanian, K. Luckow, C. Pasareanu, A. Aydin, L. Bang, T. Bultan, M. Gavrilov, T. Kahsai, R. Kersten, D. Kostyuchenko, Q.-S. Phan, Z. Zhang, and G. Karsai. ISSTAC: Integrated symbolic execution for space-time analysis of code. Under review.

[5] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *IEEE 31st International Conference on Software Engineering, ICSE '09*, pages 463–473, May 2009.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[7] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 49–60, 2016.

[8] Apache commons math. https://commons.apache.org/proper/commons-math/.

[9] A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007. Special issue on Experimental Software and Toolkits.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[13] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 395–404, 2007.

[14] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126. ACM, 1982.

[15] B. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer Berlin Heidelberg, 2008.

[16] S. Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 51–62. Springer, 2009.

[17] A. Gupta, K. L. McMillan, and Z. Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301, 2008.

[18] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.

[19] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loopiterations. *Real-Time Syst.*, 18(2/3):129–156, May 2000.

[20] Jfreechart. http://www.jfree.org/jfreechart/.

[21] Java pathfinder. http://babelfish.arc.nasa.gov/trac/jpf/.

[22] D. Kebbal and P. Sainrat. Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[24] J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 161–170, New York, NY, USA, 2013. ACM.

[25] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[26] Y.-T. S. Li, S. Malik, and B. Ehrenberg. *Performance Analysis of Real-Time Embeded Software*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[27] K. S. Luckow, C. S. Păsăreanu, and B. Thomsen. Symbolic execution and timed automata model checking for timing analysis of Java real-time systems. *EURASIP Journal on Embedded Systems*, 2015(1):1–16, 2015.

[28] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008.

[29] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a java processor. *Softw. Pract. Exper.*, 40(6):507–542, May 2010.

[30] G. Sevitsky, W. D. Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *TOOLS Europe 2001: 38th International Conference on Technology of Object-Oriented Languages and Systems, Components for Mobile Computing*, pages 85–101, 2001.

[31] R. Wilhelm. Determining bounds on execution times. In *Embedded Systems Design and Verification - Volume 1 of the Embedded Systems Handbook*, page 9. 2009.

[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem&mdash;overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[33] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In A. Gupta and S. Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 22–36. Springer Berlin Heidelberg, 2008.

[34] P. Zhang, S. G. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 43–52, 2011.