Badger: Complexity Analysis with Fuzzing and Symbolic Execution

Yannic Noller
Humboldt University of Berlin
Berlin, Germany
noller@informatik.hu-berlin.de

Rody Kersten
Synopsys, Inc.
San Francisco, USA
rody@synopsys.com

Corina S. Păsăreanu
Carnegie Mellon University Silicon Valley, NASA Ames Research Center
Moffet Field, USA
corina.s.pasareanu@nasa.gov

ABSTRACT

Hybrid testing approaches that involve fuzz testing and symbolic execution have shown promising results in achieving high code coverage, uncovering subtle errors and vulnerabilities in a variety of software applications. In this paper we describe Badger - a new hybrid approach for complexity analysis, with the goal of discovering vulnerabilities which occur when the worst-case time or space complexity of an application is significantly higher than the average case.

Badger uses fuzz testing to generate a diverse set of inputs that aim to increase not only coverage but also a resource-related *cost* associated with each path. Since fuzzing may fail to execute deep program paths due to its limited knowledge about the conditions that influence these paths, we complement the analysis with a symbolic execution, which is also customized to search for paths that increase the resource-related cost. Symbolic execution is particularly good at generating inputs that satisfy various program conditions but by itself suffers from path explosion. Therefore, Badger uses fuzzing and symbolic execution in tandem, to leverage their benefits and overcome their weaknesses.

We implemented our approach for the analysis of Java programs, based on Kelinci and Symbolic PathFinder. We evaluated Badger on Java applications, showing that our approach is significantly faster in generating worst-case executions compared to fuzzing or symbolic execution on their own.

CCS CONCEPTS

• Security and privacy → Formal methods and theory of security; **• Software and its engineering → Formal software verification**; **Software testing and debugging**;

KEYWORDS

Fuzzing, Symbolic Execution, Complexity Analysis, Denial-of-Service

ACM Reference Format:
Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213868

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213868

1 INTRODUCTION

In recent years, fuzz testing has emerged as one of the most promising testing techniques for finding correctness bugs and security vulnerabilities in software. It is used routinely by major software companies such as Microsoft [14, 37] and Google [38]. While a large fraction of the inputs generated with fuzzing may be invalid, it can be more effective in practice than more sophisticated testing techniques – such as the ones based on symbolic execution [13, 27] – due to the low computation overhead involved in fuzzing.

Fuzz testing tools, such as AFL [41] and LibFuzzer [28], have proven very successful by finding bugs and vulnerabilities in a variety of applications, ranging from image processors and web browsers to system libraries and various language interpreters. For example, AFL was instrumental in finding several of the Stagefright vulnerabilities in Android, the Shellshock related vulnerabilities in BIND as well as numerous bugs in popular applications and libraries such as OpenSSL, OpenSSH, GnuTLS, GnuPG, PHP, Apache, and IJG jpeg. In a nutshell, AFL uses genetic algorithms to *mutate* user-provided inputs using byte-level operations. These mutations are guided by coverage information obtained from running the analyzed program on the generated inputs. The interesting mutants (that are shown to increase coverage) are saved and mutated again. The process continues with the newly generated inputs, with the goal of generating a diverse set of inputs that increase the coverage of the program.

Motivated by the success of fuzz testing, we explore here the application of the technique to algorithmic complexity analysis. Characterizing the algorithmic complexity of a program has many practical applications as it enables developers to reason about their programs, understand performance bottlenecks and find opportunities for compiler optimizations. Algorithmic complexity analysis can also reveal worst-case complexity vulnerabilities, which occur when the worst-case time or space complexity of an application is significantly higher than the average case. In such situations, an attacker can mount Denial-of-Service attacks by providing inputs that trigger the worst-case behavior, thus preventing benign users to use the application.

There are several challenges in adapting fuzz testing to algorithmic complexity analysis. First, fuzz testers like AFL are designed to generate inputs that increase code coverage, while for complexity analysis one is interested in generating inputs that trigger worst case execution behavior of the programs. Furthermore, fuzzers are known to be good at finding so called *shallow* bugs but they may fail to execute deep program paths [31], i.e. paths that are guarded by specific conditions in the code. This is due to the fact that the fuzzers have little knowledge about which inputs affect which condition

in the code. On the other hand, symbolic execution techniques are particularly well suited to find such cases, but usually are much more expensive in terms of computational resources required.

We therefore propose an analysis tool that uses fuzzing and symbolic execution in tandem, to enable them to find worst case program behaviors, while addressing their limitations. Specifically, we present Badger: a framework that combines fuzzing and symbolic execution for automatically finding algorithmic complexity vulnerabilities in Java applications.

Our hybrid approach works as follows. We first run a fuzzer to generate a diverse set of inputs. For fuzzing, we build on Kelinci [20], an AFL-based fuzzer for Java programs. We modify Kelinci and AFL to add a new heuristic to account for resource-usage *costs* of program executions, meaning that the inputs generated by the fuzzer are marked as important if they obtain either an increased execution cost or new coverage. We call this tool KelinciWCA. The cost is defined in terms of number of conditions executed, actual execution time as well as user-defined costs that allow us to keep track of memory and disk usage as well as other resources of interest particular to an application.

The inputs generated by the fuzzer may cover a large set of executions but may fail to exercise deep program behavior. This can happen because of some hard-to-solve conditions that guard deep executions, as discussed above. At some user-defined point in time, the inputs are transferred to the symbolic execution side which analyzes them with the goal of producing new inputs that increase the cost and/or the coverage. These inputs are passed back to the fuzzer and the process continues until a vulnerability is found or a user-defined threshold is met.

For symbolic execution we use Symbolic PathFinder (SPF), a symbolic execution tool for Java bytecode [26]. We modified SPF by adding a mixed concrete-symbolic execution mode, similar to concolic execution [27] which allows us to *import* the inputs generated on the fuzzing side and quickly reconstruct the symbolic paths along the executions triggered by the concrete inputs. These symbolic paths are then organized in a *tree* which is analyzed with the goal of generating new inputs that expand the tree. The analysis is guided by novel *heuristics* on the SPF side that favor new branches that increase resource-costs. The newly generated inputs are passed back to the fuzzing side.

A novelty of our approach is the handling of user-dependent costs, which get translated into symbolic costs on the symbolic execution side and are handled by running a symbolic maximization procedure, to generate the worst-case inputs. This broadens the application of Badger over previous symbolic execution techniques [5, 23], which could only handle simple, concrete costs.

Scalability is achieved in two ways. First, constraint solving is turned off during concolic execution, and is used only for generating new inputs, when expanding the tree. This is done selectively, guided by the heuristics. Furthermore, the tree is saved in memory, and expanded incrementally, only when new inputs are generated.

We demonstrate how Badger is able to find a large number of inputs that trigger worst-case complexity in complex applications, and we show that it performs better than its parts (i.e. fuzzing and symbolic execution separately).

We note that we are not the first to use fuzzing and symbolic execution in a complementary manner. Tools such as Mayhem [7] and Driller [31] are prominent examples. Mayhem won first place at the recent DARPA Cyber Challenge [35], and Driller later matched those results. There are many other similar hybrid approaches, which we discuss in Section 5. However, all these previous hybrid approaches aim to increase code coverage and we believe that we are the first to explore this combination for complexity analysis. The recent work on SlowFuzz [24] explores fuzzing for worst-case analysis (WCA). Although that work addresses binaries (and not Java) and uses a different fuzzer [28], it is similar in spirit to our KelinciWCA tool. For this reason, we include in our experiments Java versions of the same (or similar) examples as in [24]. While we can not compare Badger to SlowFuzz directly, we compare Badger with KelinciWCA, which should give an indication whether a combination of SlowFuzz with symbolic execution could achieve similar benefits.

2 BACKGROUND

2.1 Fuzz Testing Java Programs with Kelinci

Kelinci is an interface to execute AFL on Java programs [20]. It adds AFL-style instrumentation to Java programs and communicates results back to a simple C program that interfaces with the AFL fuzzer. This in turn behaves as a C program that was instrumented by one of AFL's compilers.

The first step when applying Kelinci is to add AFL-style instrumentation to a Java program. AFL uses a 64 kB region of shared memory for communication with the target application. Each basic block is instrumented with code that increments a location in the shared memory bitmap corresponding to the branch made into this basic block. The Java version of this instrumentation is the following, which amounts to 12 bytecode instructions after compilation:

```
Mem.mem[id^Mem.prev_location]++;
Mem.prev_location = id >> 1;
```

In this example, the `Mem` class is the Java representation of the shared memory and also holds the (shifted) `id` of the last program location. The `id` of a basic block is a compile-time random integer, where $0 \leq id < 65536$ (the size of the shared memory bitmap). The idea is that each jump from a block $id1$ to a block $id2$ is represented by a location in the bitmap $id1 \oplus id2$. While obviously there may be multiple jumps mapping to the same bitmap location, or even multiple basic blocks which have the same `id`, such loss of precision is considered rare enough to be an acceptable trade-off for efficiency. The reason that the `id` of the previous location is shifted is that, otherwise, it would be impossible to distinguish a jump $id1 \rightarrow id2$ from a jump $id2 \rightarrow id1$. Also, tight loops would all map to the location 0, as $id \oplus id = 0$ for any id. Instrumentation is added to the program at compile time using the ASM bytecode manipulation framework [3].

Based on this lightweight instrumentation, AFL will prioritize input files that lead to newly covered branches as ancestors for the next generation of input files. Kelinci was used to find bugs in various Java applications, including Apache Commons Imaging and OpenJDK 9 [20].

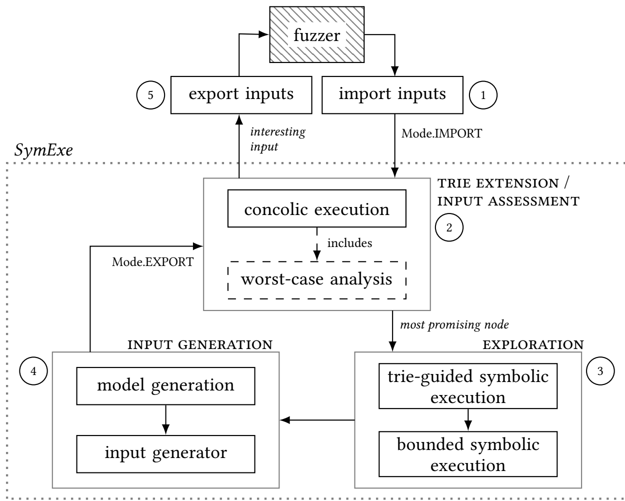


Figure 1: Badger workflow. Dashed lines represent activities that happen in parallel to the main flow.

2.2 Symbolic Execution and Symbolic PathFinder

Symbolic execution [10, 21] is a program analysis technique which executes programs on symbols in place of concrete inputs. When a decision is encountered, all branches are explored. Branch conditions are aggregated into a *path condition*, a constraint over the symbolic program inputs. Solving the path condition using an off-the-shelf solver (e.g. Z3 [11]) can detect infeasible paths as well as generate actual inputs (solutions of the constraint) that lead execution down the corresponding path. A typical use-case for symbolic execution is test-case generation [4, 6, 10]. In many cases, it can also detect faults directly [9, 14]. There are many other use-cases, including security testing [7, 25, 31] and complexity analysis [5, 23].

The research presented in this paper is based on Symbolic PathFinder (SPF), which extends the Java PathFinder framework with a symbolic execution mode [26]. This mature symbolic execution tool works on Java ByteCode and has support for most language features such as all primitive data types, strings, complex data structures, library calls, etc. It interfaces with a variety of solvers to solve constraints generated by symbolic execution.

3 APPROACH

Figure 1 shows an overview of the Badger workflow. The fuzzer block is hatched because the detailed workflow in the fuzzer is omitted in this figure. As discussed, Badger has two main components: fuzzing and symbolic execution (*SymExe* for short). Inputs are generated in an iterative manner, on both fuzzing and *SymExe* sides, and are transferred between the two components, to trigger generation of new inputs, as guided by heuristics for worst-case analysis. Specifically, the fuzzer generates and exports inputs that are shown to increase either coverage or cost on the fuzzer side; the specific costs will be discussed in detail below. These inputs are imported by *SymExe* (cf. label 1 in Figure 1). *SymExe* uses concolic execution over the imported inputs to build a partial symbolic execution tree, similar to the trie-based data structure by Yang et al. [40] (cf. label 2 in Figure 1). This trie captures the so-far explored portion of the symbolic execution paths, in which each node represents a decision in the program that includes symbolic variables. The trie is saved in memory and gets extended incrementally, whenever new inputs are imported by *SymExe*. While the trie is constructed and updated, it is also analyzed on-the-fly to compute the cost and coverage achieved for each node. This information is used by heuristics for worst-case analysis on the *SymExe* side to choose a node for further exploration in order to produce new inputs that increase coverage and/or cost.

In order to explore the chosen node, we first use a guided symbolic execution until we reach the node of interest (cf. label 3 in Figure 1). This can be done very efficiently, guided by the path in the trie that leads to that node, and with constraint solving turned off. After reaching the node of interest, we start a bounded symbolic execution (BSE) with a specified depth to explore new paths and generate corresponding new path conditions.

The collected path conditions are then solved to generate new input files (cf. label 4 in Figure 1). Since the exploration is performed on heuristically chosen nodes, and the newly generated inputs follow new paths, which were not explored before, we need to assess them, to measure their actual cost and the coverage achieved. This is done by running again concolic execution over these newly generated inputs, and updating/extending the trie in the process (cf. label 2 in Figure 1). Only the inputs that are found to lead to new interesting behavior (better cost or new coverage) are exported to the fuzzer, which will use them for its own analysis (cf. label 5 in Figure 1). The fuzzer will generate more inputs of interest, which will be imported again by *SymExe*.

We group the steps labeled with 2 to 4 in the *SymExe* box, which represents the symbolic execution component, and does not include the interaction with the fuzzer. We note that in practice, we let Badger stay in *SymExe* for a specified number of iterations, i.e. it only imports new input files from the fuzzer in intervals. The intuition is to let *SymExe* work on its own, exploring *several* promising nodes, rather than spending all its time importing information from the fuzzer. Note that similarly, the fuzzer imports new files in intervals (i.e. the havoc cycle in AFL).

Badger uses Kelinci for fuzzing and SPF for symbolic execution, which are executed in parallel. Both tools have been extended to search specifically for worst-case behavior w.r.t. a variety of cost metrics. In the following, we explain both components in detail.

3.1 Fuzzing with KelinciWCA

KelinciWCA extends Kelinci with prioritization of costly paths. Costs are collected on the Java side, then sent to AFL, which we also modified to take into account the path costs (in addition to coverage). The fuzzer maintains the current *highscore* with respect to the used cost model. When creating the next generation of inputs, the fuzzer selects ancestors from the set of inputs from the previous generation that either lead to the execution of previously unexplored program branches or to a new highscore. The chance that an input is selected from this set depends on its cost, as recorded on the Java side.

There are three cost models available:

- **Timing** is measured by counting jumps (branches) in the program. This is more precise than measuring wall clock

time, as in the latter case, there are often outliers and other inconsistencies due to, e.g., other activities on the machine. It is efficiently measured by adding the statement `Mem.jumps++` to the instrumentation trampoline, adding 4 bytecode instructions and bringing the total number to 16.

- **Memory** usage is measured by intermittent polling using a timer. The maximum consumption at any point during execution of the program on the given input is collected. Though measuring allocations using program instrumentation could in some cases be more precise, it does not take into account garbage collection, and it requires the program to determine the sizes of individual objects which is expensive and can also be inaccurate.
- **User-defined** costs can also be used. In this case, the user instruments their program with calls to the special method `Kelinci.addCost(int)`, enabling the use of arbitrary metrics like the values of variables. Moreover, it allows a relationship between input and cost about which a machine can reason. This will be used later by the symbolic execution engine to directly generate inputs with maximal costs as an optimization.

KelinciWCA inherits the ability of AFL to run in a parallel mode, which enables the synchronization with other AFL instances. After a configurable number of cycles with its own mutation operations (i.e. havoc cycles), AFL checks the other instances for interesting inputs. Since this synchronization procedure is merely based on a specific folder structure, we can pass files from our symbolic execution part to KelinciWCA easily.

3.2 Example

Before describing the *SymExe* component of Badger, we introduce an example to illustrate how the various steps work. The example is an implementation of Insertion Sort and is given in Listing 1.

```
0  public static void sort(int[] a) {
1      final int N = a.length;
2      for (int i = 1; i < N; i++) {
3          int j = i − 1;
4          int x = a[i];
5          while ((j >= 0) && (a[j] > x)) {
6              a[j + 1] = a[j];
7              j−−;
8          }
9          a[j + 1] = x;
10     }
11 }
```

Listing 1: Insertion Sort

3.3 SymExe: Symbolic Execution with Symbolic PathFinder

The *SymExe* component consists of three steps: trie extension and input assessment (cf. label 2 in Figure 1), exploration (cf. label 3 in Figure 1) and input generation (cf. label 4 in Figure 1). The following sub sections will cover all parts.

Figure 2 shows snapshots of the trie while running *SymExe* on Insertion Sort for three numbers (N=3). Note that trie nodes correspond to decisions rather than conditions in the program; multiple trie nodes may correspond to a single program condition. Since N has the concrete value 3, the only symbolic decision in our program is `a[j] > x` on line 5. Therefore, all nodes in Figure 2 except the root nodes correspond to that decision.

3.3.1 *Trie extension / input assessment.*

In this step, *SymExe* performs a concolic execution over a set of concrete inputs (which are generated either by the fuzzer or by *SymExe* itself) and updates the trie data structure that maintains a view of all the explored symbolic execution paths. This is done by adding a concolic mode to SPF that simply collects the constraints along the concrete paths (without any constraint solving) and using a listener to monitor the execution and update the trie with newly explored nodes.

Figure 2a shows the trie for the initial input of the Insertion Sort example. There are three nodes: the root node of the trie, and two decisions for `a[j] > x` on line 5, which is the only condition that depends on symbolic input. The last node (id=2) is a leaf, the last decision on the execution trace for the initial input.

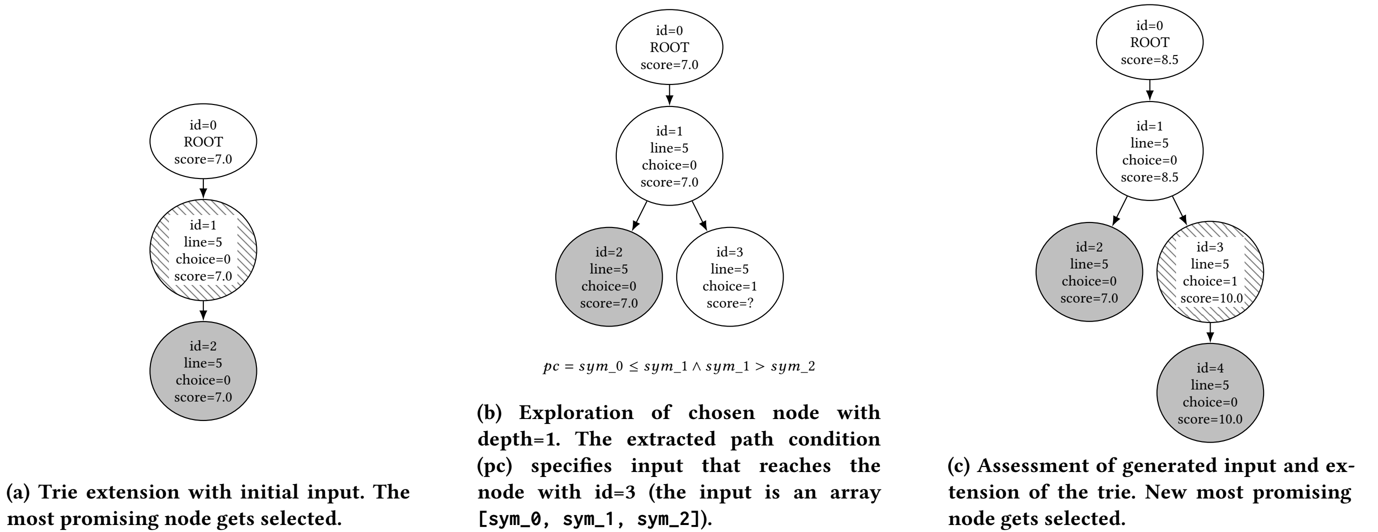
During concolic execution we perform a worst-case analysis (WCA), to compute cost and coverage information for each node in the trie. The default cost metric is timing, measured in number of jumps (branches). Alternatively, WCA can collect user-defined costs specified in the code with `Observations.addCost(int)`. A novelty of our approach is that we can handle symbolic (input-dependent) costs as described in detail below.

Each trie node gets an associated *score*, indicating how promising this node is with respect to the cost. A score reflects an estimation of total costs that may be achieved rather than costs accumulated so far along a path. The score of a leaf node is defined as the cost associated with the path that ends in that leaf node; for example, it could be the total number of branches executed to reach that node. The scores for the internal nodes are propagated up in the trie, by taking the average of children scores as the score for a parent node. For our simple example in Figure 2a all nodes get the value 7 (after importing one input) as the graph corresponds to a single path with cost 7. Furthermore, coverage information for each node is updated.

The next step is to select the most promising node for further exploration. Every node that has unexplored children is a candidate for further exploration. We choose the most promising one based on three criteria. First, nodes with unexplored choices leading to new **branch** coverage are given priority; this means that if exploring a node can potentially lead to new coverage, we consider it as a good candidate for exploration. Second, nodes are ranked based on their **score**; nodes with higher score will again be given priority. Third, nodes are ranked based on their **position** in the trie. For the latter, the tool is parametric with respect to two different heuristics:

(1) Prefer nodes **higher** in the trie. This follows the intuition of exploring a wider region of the execution tree, covering a wider input space, analogous to a breadth-first search.
(2) Prefer **lower** nodes. This follows the intuition to drill deeper in a promising direction, where high costs have been observed, analogous to depth-first search.

For the example in Figure 2a we apply the second heuristic, i.e. to prefer nodes lower in the trie. Hence, the node with id=1 is chosen as the most promising node instead of the root node. For the situation in Figure 2c (which reflects the case when the import generated by



(a) Trie extension with initial input. The most promising node gets selected.

(b) Exploration of chosen node with depth=1. The extracted path condition (pc) specifies input that reaches the node with id=3 (the input is an array `[sym_0, sym_1, sym_2]`).

(c) Assessment of generated input and extension of the trie. New most promising node gets selected.

Figure 2: Trie evolution for Insertion Sort (N=3). The most promising node for the next exploration step is presented as hatched node. Grey colored nodes denote leafs, i.e., the last decision of an execution trace.

SymExe itself is evaluated) it is clearly the node with id=3 because it is the only node with score=10.0 that can be explored.

We note that the input assessment and the trie extension can be executed in two modes: (1) import, and (2) export. The import mode is used when importing input files from KelinciWCA. There is a special case when the user-defined cost metric is used: after importing an input, *SymExe* tries to maximize the costs along the concrete path by leveraging the maximizing procedures described in Section 3.3.3. If successful, *SymExe* will immediately export the maximizing input to the fuzzer. The export mode is used when it is necessary to assess input files generated by *SymExe*. Executing them with concolic execution (and on-the-fly WCA) reveals cost values and extends the trie with new nodes. We use a conservative approach to exporting input files to KelinciWCA, because we do not want to flood the fuzzer with irrelevant inputs. Only inputs leading to new branch coverage or a new high score are exported.

3.3.2 Exploration. This step performs the actual exploration of new parts of the trie. The goal is to explore *new* branches starting from the node that was deemed most promising in the previous step (this means that the new branches will likely increase coverage or cost). This step involves a quick symbolic execution along the trie path that reaches the node, as guided by the choices encoded in the trie. There is no constraint solving needed. As soon as the node of interest is reached, we switch to bounded symbolic execution (BSE) to explore new decisions. When BSE reaches its bound we extract the path condition(s) and pass them to the next step, which will solve them to obtain new inputs.

In our example in Figure 2b the trie-guided symbolic execution phase is very short, since only one choice is made to get to the node with id=1. Next, we perform BSE with depth=1, i.e. only one step, and reach the node with id=3. The score for this new node is unknown because it is not the end of the execution trace. The extracted path condition *pc* makes it possible to generate an input value that reaches this new node. We will then need to run concolic execution again, to assess this new input, to compute the scores and update the trie (see Figure 2c).

3.3.3 Input Generation and Input-dependent Costs. In this step we generate concrete inputs by solving the path conditions from the previous step. This is done by using an off-the-shelve constraint solver. For the example, the path condition $sym_0 \leq sym_1 \wedge sym_1 > sym_2$ may be solved to generate the input `[1,1,0]`. The solution(s) are used to generate a new input file (cf. step 5 in Figure 1), which is application-dependent. Note that it could happen that some path conditions are unsatisfiable, in which case no input is generated for them.

The path condition from Figure 2b makes it possible to generate an input value that follows a path along the node with id=3. As shown in Figure 2c, this input leads to a leaf node with id=4, and to a new high score, which is back-propagated to the precedent nodes.

As mentioned, the usage of heuristics makes it necessary to assess the actual cost value of each input because it is not guaranteed that the most promising nodes actually lead to worse costs. This is done by passing the generated input back to the trie extension phase and executing in export mode as described above.

A key novelty of our approach is the incorporation of symbolic (input-dependent) costs, which require specialized techniques for input generation. The user-specified costs allow to use arbitrary variables in the program when specifying the cost, including input-dependent ones. Thus, a symbolic path may have a symbolic cost, i.e. a symbolic expression that is passed as a parameter to method `Observations.addCost(int)`. We are then interested in computing inputs that *maximize* this cost. To achieve this, we propose to use an optimization procedure in the constraint solver. In our set-up, we use Z3 [11], which allows to specify *optimization objectives*, i.e. we can ask Z3 to generate a model that maximizes a given expression. We illustrate the approach on a simple example, shown in Listing 2. If the user is interested in maximizing the value of a variable (here `sum`), then simply counting jumps or measuring

resource consumption will not be sufficient for the generation of worst-case inputs.

```
0  int sumArg(int[] a) {
1    int sum = 0;
2    for (int i=0; i < a.length; i++) {
3      if (a[i] > 0)
4        sum += a[i];
5      else
6        sum −= a[i];
7    }
8    Observations.addCost(sum);
9    return sum;
10 }
```

Listing 2: User-Defined Cost Maximization Example

To address this situation, we instrument the code with a special cost specification (line 8). When performing concolic execution over `sumArg` for concrete input values `a={1,2}`, variable `sum` at line 8 has the value sum=s_1+s_2 and the path condition is $s_1 > 0 \wedge s_2 > 0$, where s_1 and s_2 are the corresponding symbolic inputs.

We can pass an optimization term to Z3 for the specified cost expression. For example for two positive inputs the query to the solver (in SMT 2 syntax) will look like:

```
(assert (> s1 0))
(assert (> s2 0))
(maximize (+ s1 s2))
```

The retrieved model for the path condition will contain values that maximize the given expression. Assume for simplicity that the allowed range for the inputs is $[-100, 100]$. Then the maximization procedure will return $s_1 = 100, s_2 = 100$ which indicate the worst-case inputs for this path.

4 EVALUATION

In this section we present an evaluation of our implementation for Badger. In order to enable the replication of our experiments and to make our tool publicly available, we have released Badger and all evaluation artifacts on GitHub: https://github.com/isstac/badger. We evaluate our approach to answer these research questions:

RQ1: Since Badger combines fuzzing and symbolic execution, is it better than each part on their own in terms of:
  (a) Quality of the worst-case, and
  (b) Speed ?

RQ2: Is KelinciWCA better than Kelinci in terms of:
  (a) Quality of the worst-case, and
  (b) Speed ?

RQ3: Can Badger reveal worst-case vulnerabilities?

4.1 Experimental Setup

Subjects. Table 1 gives an overview of our evaluation subjects. Subject 1 to 5 are similar to the benchmarks used in SlowFuzz. Subject 6 represents an image processing application provided by DARPA, as part of STAC engagements [33]. Subject 7 represents an implementation of a smart contract for crypto-currency, translated into Java from Ethereum, where the goal is to estimate the worst-case *gas* consumption. For subject 1 to 6 we use the number of jumps as cost metric. For subject 7 we use user-defined costs that are specified directly in the code.

Table 1: Overview of the evaluation subjects.

ID	Subject	ID	Subject
1	Insertion Sort	4	Hash Table
2	Quicksort	5	Compression
3a	RegEx (fixed input)	6	Image Processor
3b	RegEx (fixed regex)	7	Smart Contract

Experiment Execution. For all subjects we ran four variations: (1) Badger, (2) KelinciWCA, (3) Kelinci and (4) *SymExe*. Success of an evolutionary fuzzing tool can depend greatly on the corpus of input files provided by the user as the first generation. For all experiments, we chose a single file with meaningless values (e.g. "Hello World") to leave the heavy lifting to the tools. Running option (1) means to execute KelinciWCA and *SymExe* in parallel. KelinciWCA starts with the initial input and *SymExe* imports the inputs from KelinciWCA like shown in Figure 1. Running option (2) and (3) means simply to execute the tools, and option (4) means to execute only *SymExe* on the initial input. We have observed in pre-experiment executions that, for our subjects, after 5 hours the compared tools reach a plateau. We therefore ran our experiments for 5 hours, 5 times (deviations from this paradigm for particular experiments are explained in the corresponding sections). We also used these pre-experiment executions to determine the best fitting heuristic for each application. Similar to Petsios et al. [24], we report slowdown in the subjects, i.e. the ratio between the costs of the observed worst case input and the initial input.

Infrastructure. All experiments were conducted on a machine running openSUSE Leap 42.3 featuring 8 Quad-Core-AMD 8384 2.7 GHz and 64 GB of memory. We used OpenJDK 1.8.0_151 and GCC 4.8.5. We configured the Java VM to use at most 10 GB of memory.

4.2 Sorting

We first evaluate our approach on two textbook examples: Insertion Sort and Quicksort. We use the implementations from JDK 1.5. Results for Insertion Sort are the averages of 5 runs. For quicksort we deviated from the usual paradigm of 5 runs and performed 10 runs, because we observed a lot of variation in the first 5 runs. For both subjects we used N=64, i.e. the input is an array of 64 integers.

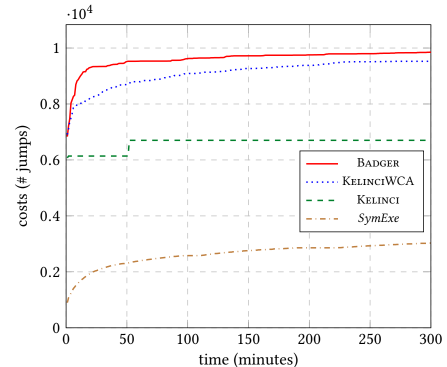


Figure 3: Results for Insertion Sort (N=64).

Results for Insertion Sort are shown in Figure 3. The score for the initial input is 509. The tools reach the following averaged final scores: Badger 9850, KelinciWCA 9533, Kelinci 6701, and *SymExe* 3025. KelinciWCA produces a slowdown of 18.73x. Badger

reaches the final score of KelinciWCA already after 61 minutes (KelinciWCA needs 219 minutes) and continues to improve to a slowdown of 19.35x after 5 hours. It is thus better in terms of both quality of the result, and speed. The symbolic execution component *SymExe* by itself performs poorly, since it is only executed on the initial input and cannot use intermediate results by KelinciWCA.

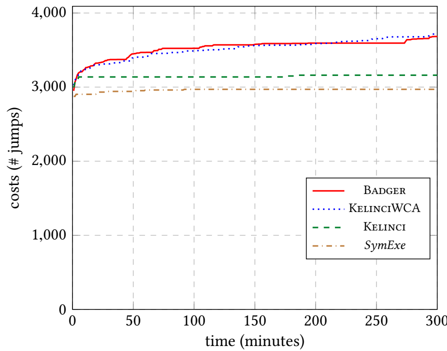


Figure 4: Results for Quicksort (N=64).

Results for Quicksort are shown in Figure 4. The score for the initial input is 2829. The tools reach the following averaged final scores: Badger 3683, KelinciWCA 3719, Kelinci 3161, and *SymExe* 2970. There is no significant difference between the results for Badger and KelinciWCA. Badger tends to be faster in generating a highscore between 20 and 150 minutes, but the final score after 300 minutes for KelinciWCA is slightly better. Badger produces a slowdown of 1.30x and KelinciWCA of 1.31x. This minor difference at the end can be explained by the randomness inherent in fuzzing; since Badger includes KelinciWCA, its results for a particular run are always at least as good. In fact for the best run of the 10 performed, we observed that Badger produced the score 4219 (1.49x), while KelinciWCA produced the score 4013 (1.42x).

4.3 Regular Expressions

The second evaluation considers regular expression matching, which can be vulnerable to so called ReDoS (Regular expression DoS) attacks. Specifically, we used the `java.util.regex` JDK package.

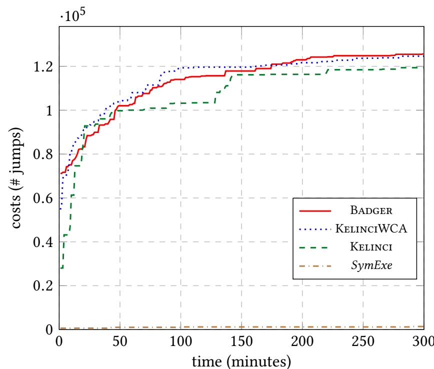


Figure 5: Results for Regular Expression (fixed regex).

We performed two experiments. In the first, we fixed the matching text and mutated the regular expression. We used the lorem ipsum filler text, and limited mutated regular expressions to 10 characters. As initial input we used the regular expression $[\backslash s \backslash S]^*$. We increased the number of experiments to 10 because we observed too much variation in the first 5 runs.

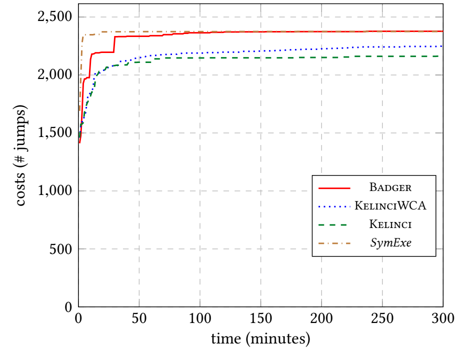


Figure 6: Results for Regular Expression (username).

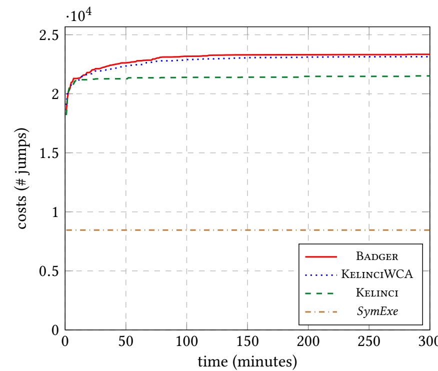


Figure 7: Results for Regular Expression (password).

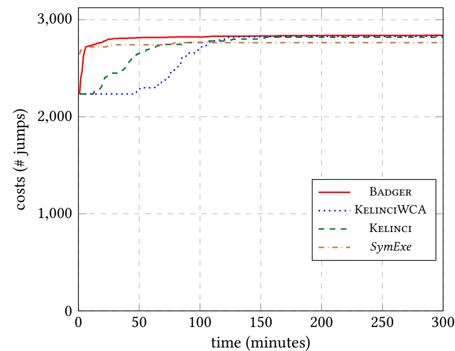


Figure 8: Results for Regular Expression (hexcolor).

Results are shown in Figure 5. The initial input score is 68101. The tools reach the following average scores: Badger 125616, KelinciWCA 124641, Kelinci 119393, and *SymExe* 1388. Badger produces a slowdown of 1.84x and KelinciWCA of 1.83x. This insignificant difference can be explained by the poor result for *SymExe*, which was not able to improve the score of the initial input.

In the second experiment, we fix the regex and mutate the text. We use the lorem ipsum as initial input again. For the regular expressions we use ten popular examples [32]. Due to space limitations, we include only the first three here, respectively matching a username, password, and hexadecimal color code:

$$\hat{}[a{-}z0{-}9_{-}]\{3,15\}\$ \tag{1}$$

$$((?=.^*\backslash d)(?=.^*[a-z])(?=.^*[A-Z])(?=.^*[@\#\$\%]).\{6,20\}) \tag{2}$$

$$\hat{}\#([A{-}Fa{-}f0{-}9]\{6\} \mid [A{-}Fa{-}f0{-}9]\{3\})\$ \tag{3}$$

Results are shown in Figures 6, 7 and 8, respectively. Remarkably, for the username regex, *SymExe* is faster than BADGER. This can be explained by the fact that the initial input already leads to relatively high costs; in general this is not the case. Additionally, *SymExe* starts working right away, while BADGER needs some time to import input files generated by KELINCIWCA (it is started with a slight delay).

Results for the password regex show that *SymExe* is not able to generate any comparable highscore, which explains why BADGER is not performing significantly better than KELINCIWCA.

For the color code regex, BADGER and KELINCIWCA both produce a slowdown of 1.27x, but BADGER finds it significantly faster. By leveraging the inputs generated by *SymExe*, BADGER reaches a cost of 2800 after 25 minutes, for which KELINCIWCA needs 133 minutes. Interestingly, KELINCI is also faster than KELINCIWCA. While statistically unlikely, this can happen due to the inherent randomness in the fuzzer.

4.4 Hash Table

The fourth evaluation subject is a hash table implementation taken from a recent DARPA engagement [33] and modified to match the hash function by SLOWFUZZ [24], which was taken from a vulnerable PHP implementation [34]. The size of the hash table is 64, each key in the hash table has a length of 8 characters, and we fill it by reading the first $64 \cdot 8$ characters from an input file. The worst-case of a hash table implementation can be triggered by generating hash collisions. Therefore, besides the normal costs, we also report the number of hash collisions. We executed the experiments 10 times because we observed too much variation in the first 5 runs.

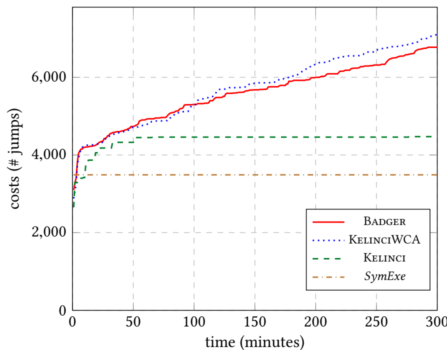


Figure 9: Results for Hash Table (N=64, key length=8).

The graph in Figure 9 shows that BADGER first performs slightly better and faster, but is passed by KELINCIWCA after 103 minutes. This is based on *SymExe*, which quickly generates a very good score, but is not able to further improve. Since we cannot report the number of collisions for the averaged plot, we looked at the best performing run for each experiment. The results correspond to 31 collisions found by BADGER, and 39 found by KELINCIWCA (the theoretical upper bound is 63). This subject shows very evidently the advantage of KELINCIWCA over KELINCI, which plateaus after 66 minutes and only finds 22 collisions.

4.5 Compression

Our fifth evaluation subject is taken from APACHE COMMONS COMPRESS. In the experiment, we BZIP2 compress files up to 250 bytes.

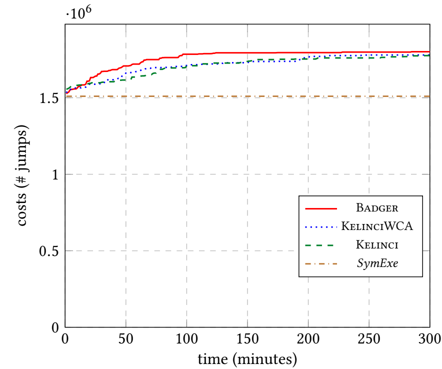


Figure 10: Results for Compression (N=250).

Results are shown in Figure 10. The score for the initial input is 1,505,039. The tools reach the following average scores: BADGER 1,800,831, KELINCIWCA 1,779,457, KELINCI 1,775,438, and *SymExe* 1,509,880. BADGER produces a slowdown of 1.20x and KELINCIWCA of 1.18x. BADGER is significantly faster. The worst case found by KELINCIWCA after 100 minutes is found by BADGER within 50 minutes.

4.6 Image Processor

Our sixth evaluation subject is an image processing application taken from a recent DARPA engagement [33]. Our analysis revealed a vulnerability related to particular pixel values in the input image causing a significantly increased runtime for the program. These pixel values trigger a particular value from a static array, which is used to determine the number of iterations in a processing loop. BADGER was able to automatically generate a JPEG image that exposes the vulnerability. For the sake of simplicity, we limited the size of images to 2x2 pixels.

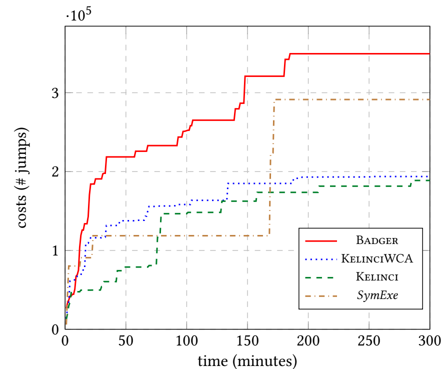


Figure 11: Results for Image Processor (2x2 JPEG).

Results are shown in Figure 11. Here we see that BADGER clearly outperforms both its components. It produces a slowdown of 40.11x, corresponding to theoretical worst-case, where all pixels have a value triggering the highest possible number of iterations in the processing loop.

4.7 Smart Contract

Our last subject is an implementation of a smart contract for cryptocurrency, where the goal is to analyse the usage of a resource

called *gas* of Ethereum software. Exceeding the allocated budget could result in loss of cryptocurrency. Therefore we consider gas as the user-defined cost in our analysis. We manually instrument the code with calls to the special methods `Kelinci.addCost(int)`, and `Observations.addCost(int)`. We executed the experiments 10 times because we observed too much variation in the first 5 runs. For our experiments we used an input array size of $N = 50$ items.

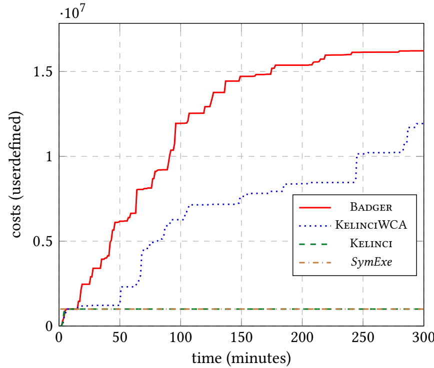


Figure 12: Results for Smart Contract (N=50).

Results are shown in Figure 12. We observed a cost of 3 for the initial input. The tools reach the following average scores: Badger 16,218,905, KelinciWCA 11,934,664, Kelinci 1,000,107, and *SymExe* 1,000,107. The increase in the cost is 5,406,301.67x for Badger and 3,978,221.33x for KelinciWCA. The cost depends on a concrete value in the input and can thus be very large (even for a short path the cost can be large if the input value is large). The initial input does not contain these large values, and hence, the cost increase is so dramatic. Badger is significantly faster than KelinciWCA, and also produces a much higher worst case cost.

4.8 Discussion

RQ1.a. Our evaluation shows that, in terms of quality of the worst case, Badger is always better than *SymExe* because eventually Badger will use the insights of its symbolic execution part. But, as mentioned earlier, Badger does not consider inputs from *SymExe* until they are imported by KelinciWCA. Additionally, the randomness from the fuzzer can cause *SymExe* to explore other paths that turn out to be less costly than those closer to the initial input. In most of our subjects, Badger also produces a better worst case than KelinciWCA. There are two cases in which KelinciWCA is slightly better than Badger: Quicksort and Hash Table. These differences are based on the randomness in the fuzzing component of our approach. Therefore, we conclude question RQ1.a with the positive answer: yes. Note that in practice, a slightly lower worst case is often more useful if it can be obtained significantly faster.

RQ1.b. Our evaluation demonstrates that Badger is significantly faster than KelinciWCA and *SymExe*, attesting a clear positive answer to RQ1.b. In most cases, *SymExe* by itself performs poorly compared to Badger. Nevertheless, there is one case where *SymExe* is able to generate a high score very fast, and Badger is not able to follow immediately: Regular Expression (username). We explain this by the random initial input, along which path *SymExe* finds a high score very quickly, whereas Badger experiences some delay in importing KelinciWCA results that also sidetrack the analysis. Note that, similar to KelinciWCA, Badger can achieve better performance through parallelization by, e.g., running multiple fuzzing instances in parallel.

RQ2.a. The evaluation shows that KelinciWCA is always better than Kelinci in terms of quality of the worst case. Although there are cases, in which Kelinci performs not bad, the single focus on coverage limits Kelinci for worst case analysis. This limitation of Kelinci was eliminated with KelinciWCA.

RQ2.b. Regarding the speed comparison between KelinciWCA and Kelinci, our evaluation illustrates that in almost all subjects KelinciWCA does not only retrieves a better worst case, but it also achieves this in shorter time. The only exception is the subject Regular Expression (hexcolor). In general, KelinciWCA is faster, but in cases where for a particular application higher coverage implies a better worst case, then Kelinci might be more efficient.

RQ3. We have shown that Badger performed well in our experiments, finding slowdowns on the subjects. Notably, Badger identified a vulnerability in Image Processor, a complex application that processes non-trivial inputs (i.e. JPEG images). Badger was able to reveal the actual worst-case by building a JPEG image, demonstrating its ability in exposing vulnerabilities in complex applications.

4.9 Threats to Validity

Internal Validity. The main threat to internal validity is the correctness of collection and analysis of experimental results. Therefore, we fully automated the process of collecting data, aggregating values and plotting graphs. Another threat to internal validity is the selection of experiment parameters, such as the heuristic worst case analysis. In order to verify this we conduced pre-experimental tests that showed the effectiveness of our selection. Additionally, in our experiments with Badger, we used one process for KelinciWCA and one for *SymExe*. In the experiments with the individual tools, each ran on a single core. This could give the combination of the tools an advantage. In future work, we plan to parallelize *SymExe*, which is expected to further improve the effectiveness of Badger, and also will enable a more fair comparison.

External Validity. The main threat to external validity is that evaluation subjects may not generalize. In order to mitigate this threat we have selected benchmarks that match existing work in the field, and added a real-world (complex) example.

Construct Validity. The main threat to construct validity is the correctness of our actual implementation. We based our implementation on SPF and Kelinci, and hence, our adaptions inherit potential incorrectness of these tools. However, face-validity shows that our evaluation results match the expected outcome.

5 RELATED WORK

SlowFuzz [24] is a fuzzer based on LibFuzzer [28] that prioritizes inputs that lead to increased execution times. The tool is similar to our KelinciWCA component, although it addresses a different programing language. Our evaluation is partially based on the SlowFuzz evaluation and not surprisingly results obtained with

KelinciWCA are similar to those for SlowFuzz. Furthermore, our experiments with Badger indicate that a combination of SlowFuzz with symbolic execution would achieve similar benefits.

Several tools explore the combination of symbolic execution with fuzzing. All these tools aim to increase coverage while our goal is to generate inputs that excersise behaviors that increase resource consumption, leading to significant technical differences.

EvoSuite [12] is a test-case generation tool for Java, based on evolutionary algorithms and dynamic symbolic execution. Unit tests are generated via random mutation, recombination and selection and are evaluated with respect to a given fitness function (typically a coverage metric). When a change in fitness is observed after mutation of a certain primitive value, the variable this value is assigned to is deemed important and dynamic symbolic execution is invoked with this variable as a symbol.

SAGE (Scalable Automated Guided Execution) [14] extends dynamic symbolic execution with a *generational search* that, instead of negating only the final condition of a complete symbolic execution, negates all conditions on the path. Solving the resulting path conditions results in a large number of new test inputs. SAGE is used extensively at Microsoft where it has been successful at finding many security-related bugs.

Mayhem [7] is a symbolic execution engine that aims to find security vulnerabilities in binaries. A *Concrete Executor Client* (CEC) explores paths concretely and performs a dynamic taint analysis. When a basic block is reached that contains tainted instructions, it is passed to the *Symbolic Executor Server* (SES) that is running in parallel. After symbolic execution, the SES instructs the CEC on a particular path to execute. Mayhem was combined with the Murphy fuzzer and won the 2016 DARPA Cyber Grand Challenge [36].

Driller [31] is another promising tool that combines the AFL fuzzer with the angr symbolic execution engine and that has achieved similar results to Mayhem. It is very similar to our approach, in that it executes a fuzzer and symbolic execution engine in parallel, combining their strengths and overcoming their weaknesses. However, while Driller is optimized for uncovering new branches, we focus on worst-case analysis.

Symbolic execution was used before for worst-case analysis [5, 23]. WISE [5] analyzes programs for small input configurations using concolic execution and attempts to *learn* a path policy that likely leads to worst-case executions at any size. This policy is then applied to programs that have larger input configurations, to *guide* the symbolic execution of the program. SPF-WCA [23] uses SPF to perform a similar analysis with more sophisticated path policies, which take into account the *history* of executions and the calling context of the analyzed procedures. In addition, SPF-WCA also uses function fitting to obtain estimates of the asymptotic complexity. Both WISE and SPF-WCA require to perform exhaustive symbolic execution for large enough input sizes to obtain good policies, which may not be feasible in practice. Furthermore, both techniques only consider one input parameter for size and usually require some manual fine tuning (e.g. for SPF-WCA manually decide the size of the history). In contrast, our technique uses a combination with fuzzing to avoid a full exhaustive symbolic execution and is fully automatic (except for the creation of drivers that are necessary for all of these approaches). Furthermore, Badger enables analysis with input-dependent costs, using a novel maximization technique. This significantly broadens the application scope of Badger w.r.t. previous techniques, which only support simple, fixed costs associated with each program path.

Load testing [42] employs symbolic execution to perform an *iterative* analysis for increasing exploration depth, with pruning of paths with low resource consumption. That work could not be used directly for finding the worst-case algorithmic behavior, since all the paths are explored up to the same depth, and therefore have the same number of steps.

Probabilistic symbolic execution is used in [8] to infer the performance distribution of a program according to given usage profiles. Although promising, the technique does not yet scale for large programs, due to the more involved probabilistic computation.

Static analysis is used in [1, 16, 17, 30] to compute conservative bounds on looping programs. In contrast to this work, our tool produces actual inputs that expose vulnerabilities, but can not provide guarantees on worst-case bounds. There is also a large body of work on worst-case execution time (WCET) analysis—in particular for real-time systems [18, 19, 22, 39]. Unlike our work, these techniques typically assume that loops have finite bounds and are independent of input, and estimate worst-case execution for specific platforms.

Profilers [2, 15, 29] are typically used for performance analysis of programs, but are inherently limited by the quality of tests used. In our work we aim to automatically generate input data triggering a diverse set of executions, including the worst-case ones.

6 CONCLUSIONS AND FUTURE WORK

We have proposed Badger, a hybrid testing approach for complexity analysis. It extends the Kelinci fuzzer with a worst-case analysis, and uses a modified version of Symbolic PathFinder to import inputs from the fuzzer, analyze them and generate new inputs that increase both coverage and execution cost. Badger can use various cost models, such as time and memory consumption, and also supports the specification of input-dependent costs. Badger was evaluated against a large set of benchmarks, demonstrating the performance and quality benefits over fuzzing and symbolic execution by themselves.

In the future, we plan to explore more heuristics for worst-case analysis on both the fuzzing and the symbolic execution side. We also plan to focus more on the symbolic execution part of Badger and conduct experiments using multiple depths during bounded symbolic execution. Additionally, we plan to explore techniques to increase scalability of the symbolic execution part by, e.g., limiting the size of the trie, which would lead to a faster exploration of the upper symbolic execution tree. Furthermore, we plan to extend our approach not only for complexity analysis, but also for a *differential* side-channel analysis of security-relevant applications.

ACKNOWLEDGMENTS

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. This work is also supported by the German Research Foundation (GR 3634/4-1 EMPRESS).

REFERENCES

[1] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* 46, 2 (February 2011), 161–203.

[2] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. 2004. Finding and Removing Performance Bottlenecks in Large Systems. In *ECOOP 2004 - Object-Oriented Programming*. 170–194.

[3] ASM. 2017. http://asm.ow2.org/. Accessed August 11, 2017.

[4] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 90–101. https://doi.org/10.1145/3092703.3092715

[5] J. Burnim, S. Juvekar, and K. Sen. 2009. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*. 463–473. https://doi.org/10.1109/ICSE.2009.5070545

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[7] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[8] Bihuan Chen, Yang Liu, and Wei Le. 2016. 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*. 49–60. https://doi.org/10.1145/2884781.2884794

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[10] L. A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Softw. Eng.* 2, 3 (May 1976), 215–222. https://doi.org/10.1109/TSE.1976.233817

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[12] J. P. Galeotti, G. Fraser, and A. Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 360–369. https://doi.org/10.1109/ISSRE.2013.6698889

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing *(PLDI '05)*. ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[14] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*. ACM, 120–126.

[16] BhargavS. Gulavani and Sumit Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Lecture Notes in Computer Science, Vol. 5123. Springer Berlin Heidelberg, 370–384. https://doi.org/10.1007/978-3-540-70545-1_35

[17] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*. Springer, 51–62.

[18] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. 2006. 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)*. IEEE Computer Society, Washington, DC, USA, 57–66. https://doi.org/10.1109/RTSS.2006.12

[19] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert Van Engelen. 2000. Supporting Timing Analysis by Automatic Bounding of LoopIterations. *Real-Time Syst.* 18, 2/3 (May 2000), 129–156. https://doi.org/10.1023/A:1008189014032

[20] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2511–2513. https://doi.org/10.1145/3133956.3138820

[21] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[22] Yau-Tsun Steven Li, Sharad Malik, and Benjamin Ehrenberg. 1998. *Performance Analysis of Real-Time Embeded Software*. Kluwer Academic Publishers, Norwell, MA, USA.

[23] Kasper Luckow, Rody Kersten, and Corina Pasareanu. 2017. Symbolic Complexity Analysis using Context-preserving Histories. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*. 58–68. https://doi.org/10.1109/ICST.2017.13

[24] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2155–2168. https://doi.org/10.1145/3133956.3134073

[25] Corina S. Păsăreanu, Quoc-Sang Phan, and Pasquale Malacaria. 2016. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Proceedings of the 2016 IEEE 29th Computer Security Foundations Symposium (CSF '16)*. IEEE Computer Society, Washington, DC, USA, 387–400. https://doi.org/10.1109/CSF.2016.34

[26] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* (2013), 1–35. https://doi.org/10.1007/s10515-013-0122-2

[27] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[28] K Serebryany. [n. d.]. Libfuzzer: A library for coverage-guided fuzz testing (within llvm). http://llvm.org/docs/LibFuzzer.html.

[29] Gary Sevitsky, Wim De Pauw, and Ravi Konuru. 2001. 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*. 85–101.

[30] Olha Shkaravska, Rody Kersten, and Marko Van Eekelen. 2010. Test-Based Inference of Polynomial Loop-Bound Functions. In *PPPJ'10: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (ACM Digital Proceedings Series)*, Andreas Krall and Hanspeter Mössenböck (Eds.). 99–108.

[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. http://www.internetsociety.org/sites/default/files/blogs-media/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[32] Website. 2012. Regular Expressions. http://www.mkyong.com/regular-expressions/10-java-regular-expression-examples-you-should-know/.

[33] Website. 2015. DARPA's Space-Time Analysis for Cybersecurity program. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity.

[34] Website. 2015. Debian Bug report log 800564 – php5: trivial hash complexity DoS attack. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=800564. Accessed Jan 28, 2018.

[35] Website. 2016. Cyber Grand Challenge Archive. http://archive.darpa.mil/cybergrandchallenge/.

[36] Website. 2016. DARPA Cyber Grand Challenge. https://www.darpa.mil/news-events/2016-08-04.

[37] Website. 2016. Microsoft Security Risk Detection. https://www.microsoft.com/en-us/security-risk-detection/.

[38] Website. 2017. OSS-Fuzz: Five months later, and rewarding projects. https://testing.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html. Accessed Jan 28, 2018.

[39] Reinhard Wilhelm. 2009. Determining Bounds on Execution Times. In *Embedded Systems Design and Verification - Volume 1 of the Embedded Systems Handbook*. 9. https://doi.org/10.1201/9781439807637.ch9

[40] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 144–154. https://doi.org/10.1145/2338965.2336771

[41] Michal Zalewski. 2017. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/. Accessed August 11, 2017.

[42] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer. 2011. Automatic generation of load tests. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. 43–52. https://doi.org/10.1109/ASE.2011.6100093