Singularity: Pattern Fuzzing for Worst Case Complexity

Jiayi Wei
The University of Texas at Austin
Austin, Texas, USA
wjydzh1@gmail.com

Jia Chen
The University of Texas at Austin
Austin, Texas, USA
grievejia@gmail.com

Yu Feng
The University of Texas at Austin
Austin, Texas, USA
yufeng@cs.utexas.edu

Kostas Ferles
The University of Texas at Austin
Austin, Texas, USA
kferles@cs.utexas.edu

Isil Dillig
The University of Texas at Austin
Austin, Texas, USA
isil@cs.utexas.edu

ABSTRACT

We describe a new blackbox complexity testing technique for determining the worst-case asymptotic complexity of a given application. The key idea is to look for an *input pattern* —rather than a concrete input— that maximizes the asymptotic resource usage of the target program. Because input patterns can be described concisely as programs in a restricted language, our method transforms the complexity testing problem to *optimal program synthesis*. In particular, we express these input patterns using a new model of computation called *Recurrent Computation Graph (RCG)* and solve the optimal synthesis problem by developing a genetic programming algorithm that operates on RCGs.

We have implemented the proposed ideas in a tool called Singularity and evaluate it on a diverse set of benchmarks. Our evaluation shows that Singularity can effectively discover the worst-case complexity of various algorithms and that it is more scalable compared to existing state-of-the-art techniques. Furthermore, our experiments also corroborate that Singularity can discover *previously unknown* performance bugs and availability vulnerabilities in real-world applications such as Google Guava and JGraphT.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; *Software testing and debugging*; • **Security and privacy** → *Denial-of-service attacks*;

KEYWORDS

Complexity testing; optimal program synthesis; fuzzing; genetic programming; performance bug; availability vulnerability

ACM Reference Format:
Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern Fuzzing for Worst Case Complexity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236039

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11.
https://doi.org/10.1145/3236024.3236039

1 INTRODUCTION

Reasoning about a program's worst-case complexity is an important problem that has many real-world applications, including performance bug detection and identification of security vulnerabilities. For instance, automated complexity analysis can identify cases where an algorithm's expected worst-case complexity does not match that of its implementation, thus indicating the presence of a performance bug. Such techniques are also useful for detecting availability vulnerabilities that allow attackers to cause denial-of-service (e.g., through algorithmic complexity attacks [5, 9, 19, 37]).

While there is a large body of literature on worst-case complexity analysis [6, 16, 17, 29], most of these techniques do not produce *worst performance inputs*, henceforth called *WPIs*, that trigger the worst-case performance behavior of the target program. Such WPIs can be used to debug performance problems and confirm the presence of security vulnerabilities. Furthermore, WPIs can shed light on the cause of worst-case executions and help programmers write suitable sanitizers to guard their code against potential DoS attacks.

In this paper, we propose a new black-box complexity testing technique to efficiently generate inputs that trigger the worst-case performance of a given program. The key insight underlying our approach is that WPIs almost always follow a *specific pattern* that can be expressed as a simple program. For instance, to trigger the worst-case performance of an insertion sort algorithm, the input array must be in reverse sorted order, which can be programmatically generated by appending larger and larger numbers to an empty list.

Based on this observation, our key insight is to transform the complexity testing problem to a *program synthesis* problem, where the goal is to find a *program* that expresses the common pattern shared by all WPIs. In particular, given a *target program* $\mathcal{P}$ whose resource usage we want to maximize, our algorithm synthesizes another program $\mathcal{G}$, called a *generator*, such that the outputs of $\mathcal{G}$ correspond precisely to the WPIs of $\mathcal{P}$. Since the common pattern underlying WPIs can often be represented using *small* generator programs, this approach allows us to discover WPIs very efficiently.

In the simplest case, a generator $\mathcal{G}$ consists of an initial input seed s together with a function f whose output is larger than its input. Since $size(f^i(s)) > size(f^j(s))$ whenever $i > j$, our method can generate arbitrarily large inputs by applying f sufficiently many times.

For instance, the input pattern ($[0], f = \lambda x.append(x, last(x))$) corresponds to an infinite sequence of inputs of the form $\{[0], [0, 0], [0, 0, 0], \ldots\}$. Thus, we can determine the worst-case complexity of the target program by using the synthesized generator to obtain many WPIs and then fitting a curve through these data points.

The problem of finding patterns that characterize WPIs corresponds to an *optimal synthesis problem*, where the goal is to synthesize a generator $\mathcal{G}$ such that the values produced by $\mathcal{G}$ maximize the target program's resource usage. Our method solves this optimal synthesis problem by performing feedback-guided optimization using genetic programming. Specifically, we represent generators using a set of DSLs called *Recurrent Computation Graphs (RCG)* that are (a) expressive enough to model most input patterns of interest and yet (b) restrictive enough to make the search space manageable. Given this representation, our method looks for an optimal RCG by applying genetic operators (e.g., mutation, crossover) to existing RCGs and biasing the search towards generators that maximize the target program's resource usage.

We have implemented these ideas in a tool called Singularity, publicly available on Github [36]. We evaluate Singularity's effectiveness on several benchmarks, including those from previous literature, real-world applications, and challenge problems from the DARPA STAC program [1]. Our experiments demonstrate Singularity's effectiveness at finding inputs that trigger the worst-case performance of various textbook algorithms whose average and worst-case complexity are different. Our experiments also demonstrate the advantages of our approach over (a) SlowFuzz, a state-of-the-art fuzzing technique for finding availability vulnerabilities, and (b) Wise, a complexity testing technique based on dynamic symbolic execution. Finally, our experiments corroborate that Singularity can find previously unknown performance bugs in widely-used Java applications such as Google Guava [15] and JGraphT [18].

In all, this paper makes the following key contributions:

- We introduce the notion of *input patterns* and show how to reduce the complexity testing problem to an optimal program synthesis problem, where the goal is to find an input pattern that maximizes the target program's resource usage.
- We introduce a new model of computation called *recurrent computation graphs (RCG)* for expressing input patterns. This RCG model can be instantiated in different ways to obtain a domain-specific language for generating inputs of many different types.
- We show how to solve the underlying optimal synthesis problem using genetic programming. Our method defines new genetic operators over RCGs and guides the search towards those input patterns that maximize resource usage.
- We implement our method in a tool called Singularity and evaluate it on a diverse set of benchmarks. Our experiments show the benefits of our approach over prior techniques and demonstrate that Singularity can discover interesting security vulnerabilities and performance bugs.

2 OVERVIEW

In this section, we present our problem definition and give a brief overview of our approach through a simple motivating example.

[1] The STAC program aims to develop program analysis techniques for finding availability and confidentiality vulnerabilities.

```
def quick_sort(xs):
  if(xs.length <= 1):
    return xs
  pivot = xs[xs.length/2]
  left, middle, right = []
  for x in xs:
    if(x==pivot):
      middle.append(x)
    elif(x<pivot):
      left.append(x)
    else:
      right.append(x)
  left = quick_sort(left)
  right = quick_sort(right)
  return concat(left, middle, right)
```

Figure 1: QuickSort with middle pivot selection

2.1 Problem Definition

Given a target program $\mathcal{P}$, our goal is to find an input pattern that triggers $\mathcal{P}$'s worst-case resource usage. As mentioned in Section 1, we represent input patterns as generator programs $\mathcal{G}$ that produce an infinite sequence of *increasingly larger* inputs for $\mathcal{P}$.

Definition 1. (Generator) Given a program $\mathcal{P}$ with signature $\tau \rightarrow \tau'$, a generator $\mathcal{G}$ for $\mathcal{P}$ is a program with signature $\texttt{unit} \rightarrow \texttt{Stream}(\tau)$. We write $\mathcal{G}_i$ to indicate the i'th element in the stream produced by $\mathcal{G}$ and require that $size(\mathcal{G}_i) > size(\mathcal{G}_j)$ whenever $i > j$.

Because our goal is to maximize the resource usage of a given program, we need a way to measure the size of an input and its corresponding resource usage. Thus, a *problem configuration* in our setting consists of a triple $(\mathcal{P}, \Sigma, \Psi)$, where $\mathcal{P}$ is the target program with signature $\tau \rightarrow \tau'$, Σ is a metric that defines the size of any value of type τ, and Ψ is a function of type $\tau \rightarrow \mathbb{R}$ that measures the resource usage of $\mathcal{P}$ on any input of type τ. In particular, we write $\Psi(s)$ to denote the resource usage of $\mathcal{P}$ on a concrete input s of type τ. We also use the notation $\mathcal{G}_{\leq n}$ to denote the largest element $\mathcal{G}_i$ such $\Sigma(\mathcal{G}_i) \leq n$.

To compare the asymptotic resource usage of two patterns, we define the following binary relation $\succ$ on a pair of generators:

Definition 2. (Relation $\succ$) A generator $\mathcal{G}$ is *asymptotically better than* another generator $\mathcal{G}'$, written $\mathcal{G} \succ \mathcal{G}'$, iff the resource usage of $\mathcal{G}$ on the target program exceeds that of $\mathcal{G}'$ for *all sufficiently large sizes*:

$$\exists \hat{n}. \forall n > \hat{n}.\ \Psi(\mathcal{G}_{\leq n}) > \Psi(\mathcal{G}'_{\leq n})$$

Given a problem configuration $(\mathcal{P}, \Sigma, \Psi)$, we now formalize our goal as the *complexity testing problem*:

Definition 3. (Complexity Testing) The goal of the *complexity testing* problem is to find an input pattern such that no other pattern is asymptotically better than it. That is, we want to find a $\mathcal{G}$ where:

$$\nexists \mathcal{G}'.\ \mathcal{G}' \succ \mathcal{G}$$

2.2 Motivating Example

We now informally describe our complexity testing technique on the simple `quickSort` example shown in Figure 1 as Python code. For concreteness, let us assume that generators are expressed in

$$
\begin{array}{lcl}
P & := & (C, \lambda x.LE) \\
E & := & IE \mid LE \\
C & := & Int \mid List \\
IE & := & Int \mid x \mid plus(IE, IE) \mid minus(IE, IE) \\
 & & \mid times(IE, IE) \mid length(LE) \\
LE & := & List \mid x \mid append(LE, E) \mid prepend(E, LE) \\
 & & \mid concat(LE, LE)
\end{array}
$$

Figure 2: A DSL where *prepend*/*append* adds an element to the head/tail of a list, respectively.

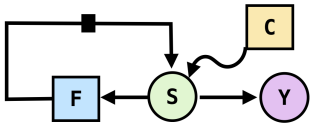

Figure 3: Output Y is obtained by repeatedly applying function F to seed value C.

a simplified DSL shown in Figure 2. Specifically, a program $\mathcal{G}$ in this language is a tuple (c, f) where c is a constant *seed* value and f is a function that operates over a list of integers. As illustrated in Figure 3, we can compute an infinite sequence of values from (c, f) by repeatedly applying f to c, where the i'th value y_i in the sequence is given by $f^i(c)$, denoting i successive applications of f to value c.

Using the DSL from Figure 2, we can express the worst-case pattern for the `quickSort` implementation from Figure 1 as follows:

$$\mathcal{G}^* = \big([0], \lambda x.append(prepend(length(x) + 1, x), length(x))\big)$$

This program produces the following sequence of inputs:

$$[0], [2, 0, 1], [4, 2, 0, 1, 3], [6, 4, 2, 0, 1, 5], \ldots$$

Observe that these inputs indeed trigger the worst-case running time of the `quickSort` implementation from Figure 1, because (a) the smallest value in each list of the sequence is the middle element, and (b) the `quicksort` implementation Figure 1 chooses the middle element as its pivot.

We now explain how Singularity finds this pattern $\mathcal{G}^*$ using genetic programming (GP). Singularity starts with a population of randomly-generated programs that conform to the context-free grammar given in Figure 2 and evaluates the fitness of each program. Since our goal is to maximize running time, the fitness function assigns a higher score to programs that take longer. For simplicity, let us assume that we evaluate running time on some particular input size, such as arrays of length 100.

Even though it is highly unlikely that the target generator $\mathcal{G}^*$ occurs in the initial population P, it might be the case that P contains several useful, albeit suboptimal, functions such as $f_1 = \lambda x.append(x, length(x))$ and $f_2 = \lambda x.prepend(length(x), x)$. These functions are useful since the desired pattern can be obtained by mixing these functions using genetic operators.

For the next iteration, the genetic programming algorithm randomly picks "fit" generators from the previous iteration. For example, the input patterns $([0], f_1)$ and $([0], f_2)$ are likely to be selected because they have higher than average resource usage. Singularity then uses these input patterns to generate a new population of candidate patterns by combining them using genetic operators, such as mutation and crossover. For example, we can obtain the following program f_3 from f_1 and f_2 using the crossover operation:

$$\lambda x.append(prepend(length(x), x), length(x))$$

In particular, crossover replaces a random sub-expression in one program with a sub-expression taken from another program. In this case, we can obtain f_3 from f_1, f_2 by substituting the sub-expression x in f_1 with the entire body of f_2. Furthermore, f_3 results in higher resource consumption compared to f_1 and f_2.

We continue the process of generating new populations and monitor both their maximal and average performance. In general, average performance will keep increasing over generations and, at some point, Singularity will generate the desired program $\mathcal{G}^*$ from $([0], f_3)$ by mutating the sub-expression $length(x)$ to $length(x) + 1$. Since $([0], f^*)$ can be used to generate an input of size 100 that achieves the maximal possible resource usage, our algorithm will eventually terminate with the desired input pattern $\mathcal{G}^*$. Observe that we can now determine the worst-case complexity of this `quicksort` implementation by measuring the running time of `quickSort` on the input values generated by $\mathcal{G}^*$ and using standard techniques to fit a curve through these data points.

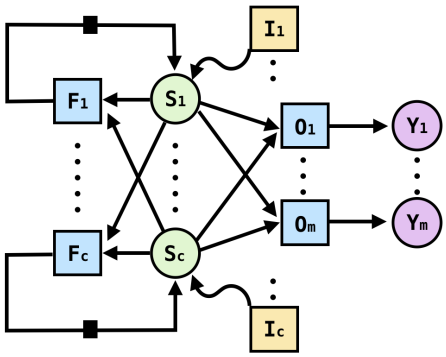

Figure 4: An RCG with c internal states and m output states.

3 RECURRENT COMPUTATION GRAPHS

In this section, we introduce *recurrent computation graphs (RCGs)* as a family of DSLs for representing generators. Intuitively, we choose RCGs as our computation model because they are expressive enough to capture most input patterns of interest that arise in practice, but they are also restrictive enough to keep the search space manageable.

Definition 4. (Recurrent Computation Graph) A *recurrent computation graph* $\mathcal{G}$ is a triple $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ where $\mathcal{I}$ is a tuple of initialization expressions, $\mathcal{F}$ is a tuple of update expressions (where $|\mathcal{I}| = |\mathcal{F}|$), and $\mathcal{O}$ is a tuple of output expressions.

Before considering the formal semantics of RCGs, we first explain them informally: An RCG $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ is a generalization of the simple computational model described in Section 2.2. As illustrated in Figure 4, instead of using one internal state, an RCG generates an infinite sequence of values by maintaining $|\mathcal{I}|$ internal states that are initialized using $\mathcal{I}$ and updated using $\mathcal{F}$. An RCG also uses an output layer $\mathcal{O}$ to transform its internal states before outputting them. This decoupling allows the number of internal states to be

$$
\begin{aligned}
s_i[0] &= [\![\mathcal{I}_i]\!] \\
s_i[t+1] &= [\![\mathcal{F}_i]\!][s_1 \mapsto s_1[t], \ldots, s_c \mapsto s_c[t]] \\
y_j[t] &= [\![\mathcal{O}_j]\!][s_1 \mapsto s_1[t], \ldots, s_c \mapsto s_c[t]] \\
&\quad \text{where } 1 \le i \le c = |\mathcal{I}| \text{ and } 1 \le j \le m = |\mathcal{O}| \\
[\![(\mathcal{I}, \mathcal{F}, \mathcal{O})]\!] &= \left[(y_1[t], \ldots, y_m(t)) \mid t \in [0, \infty]\right]
\end{aligned}
$$

Figure 5: Recurrent computation graph semantics

different from the number of arguments that the target program takes. As before, we can generate the k'th value in the infinite sequence by updating the internal states exactly k times.

RCG semantics. More formally, the semantics of an RCG $(\mathcal{I}, \mathcal{F}, \mathcal{O})$ is given by the rules shown in Figure 5. Here, $s_i[t]$ represents the i'th internal state at time step t, and $y_i[t]$ corresponds to the i'th output value at time t. As shown in Figure 5, $s_i[0]$ is computed using the i'th initialization expression in $\mathcal{I}$, and $s_i[t+1]$ is obtained from $(s_1[t], \ldots, s_c[t])$ by applying the update function $\mathcal{F}_i$. Finally, $y_j[t]$ is obtained from the internal state at time t by applying the output expression $\mathcal{O}_j$ to $(s_1[t], \ldots, s_c[t])$. The semantics of the RCG is then given by the infinite sequence of values $(y_1[t], \ldots, y_m[t])$ for $t = 0, 1, 2, \ldots$ Given an RCG $\mathcal{G}$ and a value y, we say that y is in the language of $\mathcal{G}$, written $\mathcal{L}(\mathcal{G})$, if $y = (y_1[t], \ldots, y_m[t])$ for some time step t.

RCG expressions. Our definition of recurrent computation graphs intentionally does not fix the expression language over which $\mathcal{I}, \mathcal{F}, \mathcal{O}$ are specified. To maximize the flexibility of our approach, RCGs are parametrized by a set of components C over which the initialization, update, and output expressions are constructed. Recall that both $\mathcal{F}$ and $\mathcal{O}$ are functions, and their arguments correspond to the RCG's internal states. Hence, expressions e for $\mathcal{F}$ and $\mathcal{O}$ can be generated according to the following grammar:

$$e \;:=\; s_i \;\mid\; c \;\mid\; f(e_1, \ldots, e_k)$$

where s_i represents the i'th internal state, c is a constant value, and $f \in C$ is a function of arity k. Since initialization expressions are required to be constants, *init* follows a similar grammar except that we do not allow initialization expressions to refer to the RCG's internal states.

Example 1. The `quickSort` pattern from Section 2.2 can be expressed as the following 2-state RCG using the components `plus`, `append`, `prepend`, `inc`, as well as integer constants $\{0, 1, 2\}$.

$$
\begin{aligned}
\mathcal{I} &= (1, [0]) \\
\mathcal{F} &= (\text{plus}(s_1, 2), \text{append}(\text{prepend}(\text{inc}(s_1), s_2), s_1)) \\
\mathcal{O} &= s_2
\end{aligned}
$$

The first few iterations of the pattern's evaluation are shown below, where we use $(\triangleright), (\triangleleft), (+)$ to denote *append*, *prepend*, and *plus* respectively:

$$
\begin{aligned}
s_1[0] &= 1 & s_2[0] &= [0] \\
s_1[1] &= 1 + 2 = 3 & s_2[1] &= (\text{inc}(1) \triangleleft [0]) \triangleright 1 = [2, 0, 1] \\
s_1[2] &= 3 + 2 = 5 & s_2[2] &= (\text{inc}(3) \triangleleft [2, 0, 1]) \triangleright 3)) = [4, 2, 0, 1, 3]
\end{aligned}
$$

In the previous example, the output state was exactly the same as one of the internal states. However, as illustrated by the following example, this is not always the case.

Example 2. Consider the following sequence of inputs: [ ], [1, 1], [1, 2, 1, 2], [1, 2, 3, 1, 2, 3], [1, 2, 3, 4, 1, 2, 3, 4], ... This input pattern can be represented using the following RCG:

$$
\begin{aligned}
\mathcal{I} &= (0, [\,]) \\
\mathcal{F} &= (\text{plus}(s_1, 1), \text{append}(s_2, s_1)) \\
\mathcal{O} &= \text{concat}(s_2, s_2)
\end{aligned}
$$

The output here is obtained by concatenating two copies of the input state s_2; however, there is no simple way to express this pattern without distinguishing between internal and output states.

4 COMPLEXITY TESTING AS DISCRETE OPTIMIZATION

In this section, we formulate the complexity testing problem introduced in Section 2.1 as an *optimal program synthesis problem* [2]. Towards this goal, we first introduce the concept of a *measurement model* for assigning scores to recurrent computation graphs:

Definition 5. (Ideal measurement model) Given an RCG $\mathcal{G}$, an *ideal measurement model* $\mathcal{M}$ maps $\mathcal{G}$ to a numeric value such that:

$$\forall \mathcal{G}, \mathcal{G}'.\ (\mathcal{G} \succ \mathcal{G}' \rightarrow \mathcal{M}(\mathcal{G}) > \mathcal{M}(\mathcal{G}')) \tag{4.1}$$

In other words, an ideal measurement model $\mathcal{M}$ assigns a higher score to $\mathcal{G}$ compared to $\mathcal{G}'$ if $\mathcal{G}$ induces asymptotically worse behavior of the target program compared to $\mathcal{G}'$. Using this notion, we now formulate complexity testing in terms of the following pattern optimization problem:

Definition 6. (Pattern Optimization) Given an ideal measurement model $\mathcal{M}$, the *pattern optimization* problem is to find an RCG that maximizes $\mathcal{M}$, i.e., find the solution of:

$$\underset{\mathcal{G}}{\text{argmax}}\ \mathcal{M}(\mathcal{G}) \tag{4.2}$$

Because RCGs correspond to programs, Definition 6 is a form of optimal program synthesis problem, where the goal is to maximize asymptotic resource usage. The following theorem states that the pattern optimization problem is equivalent to our definition of the complexity testing problem from Section 2.1:

Theorem 4.1. *Eqn. 4.2 gives a solution to Definition 3.*

Proof: Suppose pattern $\mathcal{G}$ satisfies Eqn. 4.2. If $\mathcal{G}$ is not a solution to Definition 3, then we have some $\mathcal{G}'$ such that $\mathcal{G}' \succ \mathcal{G}$. Using Eqn. 4.1, we know that $\mathcal{M}(\mathcal{G}') > \mathcal{M}(\mathcal{G})$, which means $\mathcal{G}$ is not the solution to Eqn. 4.2 (i.e., contradiction). □

Theorem 4.1 is useful because it allows us to turn the complexity testing problem into a discrete optimization problem, assuming that we have access to an ideal measurement model $\mathcal{M}$. However, due to the black-box nature of our approach, $\mathcal{M}$ is difficult to obtain in practice. In particular, the ideal measurement model requires reasoning about the asymptotic resource usage of the program on all inputs of a given shape, but this is clearly a very difficult

[2] In optimal program synthesis [2] the goal is to synthesize a program that not only satisfies the specification but also maximizes the value of some objective function

static analysis problem. Thus, as a proxy to this idealized metric, we instead estimate the quality of an input pattern by using an *empirical* measurement model $\mathcal{M}^{\hat{n}}$. Specifically, a measurement model $\mathcal{M}^{\hat{n}}$ evaluates the quality of a generator $\mathcal{G}$ by running the input program $\mathcal{P}$ on inputs up to size $\hat{n}$. In the remainder of this paper, we use the following empirical model as a proxy for Definition 5:

Definition 7. (Empirical Measurement Model) Our *empirical measurement model*, denoted $\mathcal{M}^{\hat{n}}$, evaluates an input pattern by returning the maximum resource usage among all inputs whose size does not exceed bound $\hat{n}$. More formally:

$$\mathcal{M}^{\hat{n}}(\mathcal{G}) = \max_{x \in \mathcal{L}(\mathcal{G}) \wedge \Sigma(x) \leq \hat{n}} \Psi(x) \tag{4.3}$$

The following theorem states the conditions under which $\mathcal{M}^{\hat{n}}$ is a good approximation of the ideal model:

Theorem 4.2. *$\mathcal{M}^{\hat{n}}$ is an ideal measurement model (i.e., satisfies equation 4.1) if $\hat{n}$ is sufficiently large and we have:*

$$\lim_{n \to \infty} \Psi(\mathcal{G}_{\leq n}) = \infty$$

Proof: We show that $\mathcal{G} \succ \mathcal{G}'$ implies $\mathcal{M}^{\hat{n}}(\mathcal{G}) > \mathcal{M}^{\hat{n}}(\mathcal{G}')$ under the conditions stated in the theorem. Suppose $\mathcal{G} \succ \mathcal{G}'$. From Definition 2, this means there exists n_1 such that $\forall n \geq n_1.\ \Psi(\mathcal{G}_{\leq n}) > \Psi(\mathcal{G}'_{\leq n})$. Because we assume all patterns' resource usage increase to infinity as the input size grows, we can show that there exists some n_2 such that $\forall n \geq n_2. \mathcal{M}^n(\mathcal{G}) = \Psi(\mathcal{G}_{\leq n})$ and $\mathcal{M}^n(\mathcal{G}') = \Psi(\mathcal{G}'_{\leq n})$ using Eqn. 4.3. Thus, for $\hat{n} \geq \max(n_1, n_2)$, we have $\mathcal{M}^{\hat{n}}(\mathcal{G}) > \mathcal{M}^{\hat{n}}(\mathcal{G}')$. □

5 FINDING OPTIMAL RCG USING GP

We now describe a genetic programming (GP) algorithm for solving the discrete optimization problem from Section 4. We first present the top-level algorithm and then explain its subroutines.

5.1 Algorithm Overview

Our pattern maximization algorithm is summarized in Algorithm 1 and follows the typical structure of genetic programming. Specifically, we start with a randomly-generated initial population of RCGs (lines 2-3) and repeatedly create a new population by combining the fittest individuals from the old population.

To create a new population *pop'*, we create m new RCGs by combining individuals from the existing population *pop* — this corresponds to the `for` loop at lines 6-14. A new individual $\mathcal{G}$ is created by randomly choosing a genetic operator *op* (line 7) and combining *op.arity* individuals from the current population. While there are several different techniques that can be used to select individuals from the population, our algorithm uses the so-called *deterministic tournament method* (lines 8-9). Specifically, we sample K RCGs and choose the RCG with the best fitness as the winner. [3]

Given the new RCG $\mathcal{G}$ created at line 10, we evaluate $\mathcal{G}$'s fitness (line 11) using a fitness function that we discuss in more detail in Section 5.3. If $\mathcal{G}$ is fitter than the previously fittest RCG, we then update *best* to be $\mathcal{G}$. The algorithm terminates with solution *best* if there has been no fitness improvement on *best* for many generations (line 4).

[3] K is a hyper-parameter called *tournament size* and controls the *evolution pressure* of the GP process: When K is set to 1, there is no evolution pressure and all individuals from the population, regardless of their fitness, have the same chance to be picked by the tournament method; hence, in this case, GP degenerates to random search. When K is set to the size of the whole population, only the best individual of each population can be selected to participate in the creation of new individuals.

Algorithm 1 Pattern Maximization using GP

Input: *gpOps* - the set of generic operators to use
Input: m - population size
Input: K - tournament size
Input: $\hat{n}$ - size bound for performance measurement.
Input: μ, α - hyper-parameters used for calculating fitness
Output: the pattern with the highest fitness score so far

```
1:  procedure FindOptimalRCG(gpOps, m, K, n̂, μ, α)
2:      pop ← initPopulation(m)
3:      best ← findBest(pop)
4:      while not converged() do
5:          pop' ← ∅
6:          for i from 1 to m do
7:              op ← randomPick(gpOps)
8:              for j from 1 to op.arity do
9:                  args_j ← tournament(pop, K)
10:             G ← op(args)
11:             G.fitness ← M^n̂(G) · e^{−(size(G)/μ)^4} · α^{cost(G)}
12:             if G.fitness > best.fitness then
13:                 best ← G
14:             pop' ← pop' ∪ {G}
15:         pop ← pop'
16:     return best
```

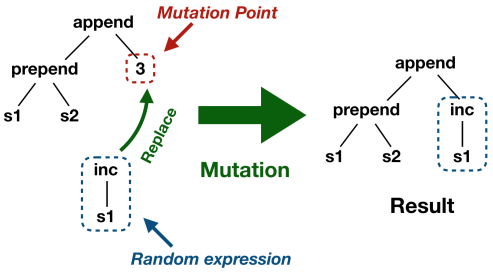

Figure 6: Mutation operator

5.2 Genetic Operators

We now describe the genetic operators used in Algorithm 1.

Mutation operator. The mutation operator is used to maintain diversity from one generation to the next and prevents the algorithm from converging on a local – rather than global – optimum. It creates an RCG $\mathcal{G}'$ from an existing RCG $\mathcal{G}$ by applying modifications to a node in the abstract-syntax tree (AST) representation of $\mathcal{G}$. Specifically, we first randomly choose an initialization, update, or output expression e and then select a random node n, called the *mutation point*, in e. Our mutation operator then replaces the sub-tree T rooted at n with a randomly generated AST with the same type as T. Figure 6 illustrates this process.

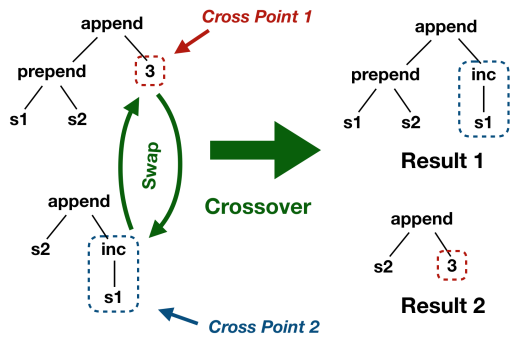

Figure 7: Crossover operator

Crossover operator. The crossover operator is used to combine existing members of a population into new individuals. Specifically, given RCGs $\mathcal{G}_1$ and $\mathcal{G}_2$, we choose a mutation point n_1 of type τ in $\mathcal{G}_1$ as well as another mutation point n_2 of the same type τ from $\mathcal{G}_2$. We then create two new RCGs by swapping the sub-trees rooted at n_1 and n_2 and randomly pick one of the two new RCGs. The crossover operation is illustrated in Figure 7.

Reproduction operator. The reproduction operator is just an identity function – it simply copies the selected individual into the next generation. Reproduction is used to maintain stability between generations by preserving the fittest individuals.

ConstFold operator. The *ConstFold* operator is similar to reproduction except that it also performs light-weight constant folding on the AST. Using *ConstFold* allows continuous evolution of constants used in the RCGs without growing total AST size.

5.3 Fitness Function

Since our goal is to find an RCG that maximizes the target program's resource usage, the simplest implementation of the fitness function simply uses the measurement model $\mathcal{M}$. However, as standard in genetic programming, the fitness function does not have to be exactly the same as the optimization objective. We design our fitness function to have the following three properties:

(1) It should be consistent with the measurement model $\mathcal{M}$, meaning that $\mathcal{G}$ is considered fitter than $\mathcal{G}'$ if $\mathcal{M}(\mathcal{G}) > \mathcal{M}(\mathcal{G}')$.
(2) It should prevent individuals from evolving to unboundedly large programs by penalizing RCGs with very large AST size.
(3) When two RCGs have similar size and resource usage, it should use the Occam's razor principle to prefer the simpler one.

Based on these criteria, our fitness function F is defined as:

$$F(\mathcal{G}) = \mathcal{M}^{\hat{n}}(\mathcal{G}) \cdot e^{-(size(\mathcal{G})/\mu)^4} \cdot \alpha^{cost(\mathcal{G})}$$

where *size* measures the total AST size of $\mathcal{G}$, and *cost* is a measure of the complexity of the RCG [4]. Both μ and α are tunable hyper-parameters. Specifically, μ is used for bloat control: If the AST size of $\mathcal{G}$ is smaller than μ, then $e^{-(size(\mathcal{G})/\mu)^4}$ is close to 1; but, when $size(\mathcal{G}) > \mu$, the fitness quickly decays to 0. The hyper-parameter α must be chosen as a value less than 1 and determines the penalty factor associated with complexity.

[4] We define complexity in terms of the constants used in the RCG. Intuitively, the larger the constants used in the RCG, the higher the cost.

6 IMPLEMENTATION

We have implemented the proposed method in a tool called Singularity, which consists of approximately 6,000 lines of Scala code, and made it publicly avaibale on Github [36]. In what follows, we discuss important design and implementation choices underlying Singularity.

Resource usage measurement. Recall that our problem definition and fitness evaluation function use a resource measurement function Ψ. We implement Ψ by counting the number of executed instructions rather than measuring absolute running time, as the latter strategy is too noisy due to factors such as cache warm-up, context switching, garbage collection etc.

To measure the executed number of instructions, we perform static instrumentation using the Soot framework [33] for Java programs and the LLVM framework [20] for C/C++ programs. In more detail, we initialize an integer counter when the application starts and increment it by one after each instruction. Our implementation also provides a lighter-weight version of this instrumentation that only increments the counter at method entry points and loop headers. In practice, we found this alternative strategy to work quite well, as it strikes a good balance between precision and overhead. Unless stated otherwise, all of our benchmarks are instrumented using this lightweight strategy.

RCG components. Recall from Section 3 that our recurrent computation graphs are parameterized by a set of components that are used to construct expressions. Our implementation comes with a library of such components for most built-in types and collections. For instance, the component library for integers include methods such as `inc`, `dec`, `plus`, `minus`, `times`, `mod` etc. Similarly, for lists, we have generic components such as `append`, `prepend`, `access`, `concat`, `length` and so forth. For graphs, we have components that represent empty graphs as well as operations that add nodes and edges (see Table 4). Since our framework is fully extendable, the user can apply Singularity to programs that take custom data types τ by providing new components that operate over τ.

Parameter tuning. As mentioned earlier, genetic programming algorithms have many tunable parameters such as population size, tournament size, threshold μ and cost penalty factor α used in the fitness function etc. Unfortunately, these parameters are often hard to configure manually due to the complex dynamics of genetic programming and the intricate interaction between different parameters. To address this problem, we developed an automatic parameter generator which samples these parameters from a joint distribution. When we run Singularity multiple times on a problem, we always use different parameter sets sampled from this joint distribution. In our experience, this strategy increases the likelihood that Singularity will find the desired worst-case pattern.

7 EVALUATION

To evaluate the usefulness of Singularity, we design a series of experiments that are intended to address the following questions:

(1) Is Singularity useful for revealing the worst-case complexity of a given program?
(2) How does Singularity compare with state-of-the-art testing tools that address the same problem?

Table 1: Evaluation on textbook algorithms.

Algorithm Name	Best Case	Worst Case	Found Worst?
Optimized Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	✓
Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
Optimized Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
3-way Quick Sort	$\Theta(n \log n)$	$\Theta(n^2)$	✓
Sequential Search	$\Theta(1)$	$\Theta(n)$	✓
Binary Search	$\Theta(1)$	$\Theta(\log n)$	✓
Binary Search Tree Lookup	$\Theta(1)$	$\Theta(n)$	✓
Red-Black Tree Lookup	$\Theta(1)$	$\Theta(\log n)$	✓
Separate Chain Hash Lookup	$\Theta(1)$	$\Theta(n)$	✓
Linear Probing Hash Lookup	$\Theta(1)$	$\Theta(n)$	✓
NFA Regex Match	$\Theta(m+n)$	$\Theta(mn)$	✓
Booyer-Moore Substring	$\Theta(m+n)$	$\Theta(mn)$	✓
Prim Minimum Spanning Tree	$\Theta(V+E)$	$\Theta(E \log V)$	✓
Bellman-Ford Shortest Path	$\Theta(1)$	$\Theta(V(V+E))$	✓
Dijkstra Shortest Path	$\Theta(1)$	$\Theta(E \log V)$	✓
Alternating Path Bipartite	$\Theta(V)$	$\Theta(V(V+E))$	✓
Hopcroft-Karp Bipartite	$\Theta(V)$	$\Theta(E\sqrt{V})$	✗

(3) Is Singularity useful for detecting algorithmic complexity vulnerabilities and performance bugs in real world systems?

Unless stated otherwise, experiments are conducted on an Intel Xeon(R) computer with an E5-1620 v3 CPU and 64G of memory running on Ubuntu 16.04.

7.1 Asymptotic Bound Analysis

In this section, we evaluate Singularity on standard algorithms, such as sorting, searching, graph algorithms, and string matching, that are taken from a widely-used algorithms textbook by Sedgewick and Wayne [28]. The goal of this experiment is to determine whether Singularity can identify the worst-case asymptotic complexity of these algorithms.

To ensure the benchmarks are nontrivial, we only focus on algorithms whose worst-case running time is known to us and is different from their best cases. Based on these criteria, we obtain a total of 17 algorithms. For each of them, we run Singularity for a total time of 3 hours and restart fuzzing with a different random seed whenever the fitness has no improvement for more than 150 generations. Finally, we determine worst-case complexities by using input patterns that maximize resource usage at size $\hat{n} = 250$.

The results of this experiment are summarized in Table 1. The first three columns of this table provide the name of the algorithm along with its corresponding best-case and worst-case asymptotic performance, and the final column shows whether Singularity is able to trigger the expected worst-case complexity. To determine whether a pattern's worst-case complexity has been found, we measure its performance at different input sizes and try to fit a linear relationship between the theoretical worst-case performance and the actual performance. If the data show a linear trend and the R^2 metric is greater than 0.95, we conclude that Singularity is able to generate inputs with the desired worst-case complexity.

As we can see from this table, Singularity can trigger the worst-case behavior in 16 of the 17 cases. For the Hopcroft-Karp bipartite matching algorithm, the inputs generated by Singularity trigger $O(V+E)$ complexity rather than the expected $O(E\sqrt{V})$ complexity because the worst-case pattern cannot be represented using our standard set of graph components listed in Table 4.

7.2 Comparison Against Wise

To explore how Singularity compares against other complexity testing techniques, we perform a comparison between Singularity and Wise [4]. Unlike Singularity, Wise is a white-box testing tool based on dynamic symbolic execution. Specifically, Wise proceeds in two phases: In the first phase, it performs exhaustive search on small inputs to learn so-called *branch policy generators*, which exercise worst-case execution paths. In the second phase, Wise uses the output of the first phase to prune program paths that do not conform to the learnt branch policy generator.

We perform this experiment on the benchmarks that are used for evaluating Wise [4]. We give both tools a time limit of three hours and compare the performance of each benchmark on the inputs generated by Singularity and Wise. Specifically, we "train" Wise on the same training size reported in their paper [4] and use both tools to generate inputs up to size n for $n \in \{30, 500, 1000\}$. Specifically, we use $n = 30$ to match the value used in the original Wise paper. We also report $n = 500$ and $n = 1000$ to demonstrate the advantages of our approach over Wise.

The results of this experiment are summarized in Table 2. Here, the symbol ✗ indicates that the tool failed to generate any inputs within the 3 hour time-limit. Otherwise, the number indicates the worst-case performance (in terms of instruction count) of the algorithm on inputs generated by each tool.

The main take-away from this experiment is that Singularity and Wise trigger roughly the same performance behavior in all cases where Wise does not time out (i.e., generates an input within the 3-hour time limit). However, as we increase the value of n, Wise fails to generate inputs on more and more benchmarks. In particular, Wise can trigger the worst-case behavior on 8 out of the 9 benchmarks for $n = 30$, but this number drops to 6 for $n = 500$ and to 3 for $n = 1000$. Specifically, Wise fails to generate any inputs for large values of n because all paths explored by the concolic execution engine within the time limit are pruned by the generator, meaning that Wise fails to find any inputs that can trigger worst-case behavior. In contrast, by looking for input patterns rather than concrete inputs, Singularity can scale to much larger values of n.

7.3 Comparison Against SlowFuzz

In our next experiment, we compare Singularity against SlowFuzz [26], a state-of-the-art fuzzing tool for finding availability vulnerabilities. Similar to our approach, SlowFuzz performs resource-usage-guided evolutionary search but generates *concrete inputs*, as opposed to *input patterns*, that maximize resource usage.

We compare Singularity with SlowFuzz in terms of scalability and the quality of the generated inputs. Similar to Section 7.2, we assess scalability by running each tool on increasing input sizes ranging from 64 bytes to 2K bytes. To evaluate the quality of the results, we run both tools 30 times with a 2-hour time limit for each run and compare the *largest* resource usage obtained by each tool. To reduce the time required to perform this experiment, we run both tools on an HPC cluster with Intel Xeon Phi 7250 CPU (68 cores at 1.4GHz) and 96G RAM running CentOS 6.3.

Table 2: Worst-case number of instructions executed on the WISE benchmarks

Benchmark	size = 30		size = 500		size = 1000	
	WISE	SINGULARITY	WISE	SINGULARITY	WISE	SINGULARITY
SortedList insert	262	262	4022	4023	✗	8023
Heap insert (JDK 1.5)	160	160	280	281	310	311
RedBlackTree insert	221	221	403	404	455	456
QuickSort (JDK 1.5)	3,522	3,638	✗	470,232	✗	1,815,732
BinarySearchTree insert	205	212	3,495	3,510	✗	7,010
MergeSort (JDK 1.5)	3,922	3,954	113,771	107,601	251,039	238,999
Bellman-Ford (adjacency matrix)	303,152	333,357	✗	1.94×10^9	✗	1.55×10^{10}
Dijkstra (adjacency matrix)	12,363	12,620	3,496,003	3,510,006	✗	1.40×10^7
Traveling Salesman	✗	$> 10^{12}$	✗	$> 10^{12}$	✗	$> 10^{12}$

Symbol ✗ indicates that the tool fails to produce any inputs within 3 hour.

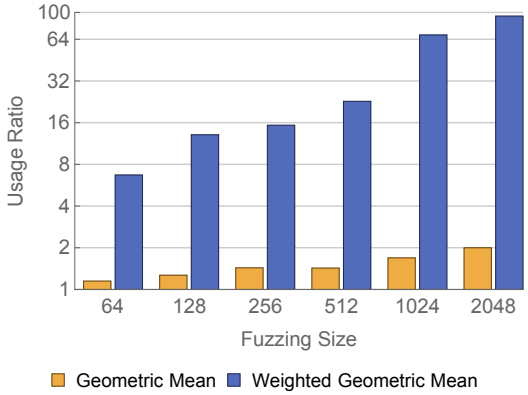

Figure 8: Comparison against SLOWFUZZ. The usage ratio represents the ratio between the worst-case resource usage found by SINGULARITY and by SLOWFUZZ.

The benchmarks for this experiment include those reported in the SLOWFUZZ paper [26], which consist of several sorting algorithms, a hash table implementation from PHP, 19 regular expression matching problems, and a zip utility from the `bzip2` application. We do not use the `bzip2` example in our evaluation since the vulnerability is triggered only when certain bits in the input file header are set; hence, this benchmark is not related to the input pattern generation problem addressed in this paper.

Since this experiment involves 27 benchmarks and 6 different input sizes, we report the aggregate results for each size. For each benchmark b and size n, we use inputs I and I' generated by SINGULARITY and SLOWFUZZ to compute the usage ratio r_b^n:

$$r_b^n = \frac{\Psi_b(I)}{\Psi_b(I')}$$

where $\Psi_b(I)$ denotes the running time (in terms of instruction count) of benchmark b on input I. Observe that $r_b^n > 1$ indicates that inputs generated by SINGULARITY take longer to run.

To aggregate over all benchmarks for each input size, we consider two different metrics:

- *Geometric mean:* For each input size s and benchmarks $b_1, \ldots, b_k$, we compute the geometric mean, denoted $GM(r_{b_1}^n, \ldots, r_{b_k}^n)$, of ratios $r_{b_1}^n, \ldots, r_{b_k}^n$.
- *Weighted geometric mean:* Since the usage ratio r_b^n is close to 1 for about half of the benchmarks, the geometric mean does not convey the full story. Instead, we want to assign a small weight to cases where both tools have similar performance, and assign a larger weight when there is a significant performance difference. Hence, we also compute the following *weighted geometric mean* [5]:

$$WGM(r_{b_1}^n, \ldots, r_{b_k}^n) = \exp\left(\frac{\sum_{i=1}^k \ln(r_{b_i}^n)^3}{\sum_{i=1}^k \ln(r_{b_i}^n)^2}\right)$$

The results of this comparison are summarized in Figure 8. We can observe two main trends based on this figure: First, SINGULARITY is able to generate inputs that cause the applications to run significantly longer within the time frame, showing that SINGULARITY is more efficient than SLOWFUZZ in terms of fuzzing efficiency. Second, the performance ratios grow as n increases, showing that SINGULARITY scales better compared to SLOWFUZZ. Hence, these results highlight the scalability advantage of pattern fuzzing over concrete input fuzzing.

7.4 Availability Vulnerability Detection

To demonstrate that SINGULARITY can generate inputs that exercise non-trivial algorithmic complexity vulnerabilities, we evaluate SINGULARITY on ten benchmarks taken from the DARPA STAC program. Specifically, we choose exactly those benchmarks that (a) exhibit an availability vulnerability, and (b) where it is possible to construct an exploit using a malicious input pattern.

In more detail, each STAC benchmark is a Java application containing between 500 to 20,000 lines of code. Furthermore, each benchmark comes with a pre-defined input budget b and a target running time t, and the goal is to craft an attack vector that causes the running time of the application to exceed t using an input of size at most b. Table 3 provides more detailed information about these STAC benchmarks.

To perform this experiment, we run SINGULARITY for a total of 3 hours on each benchmark. By default, we use a fuzzing size of 1KB, unless the specified input budget b is smaller.

As summarized by the results in Table 3, SINGULARITY is able to generate the desired attack vector for 8 out of these 10 benchmarks. To understand the limitations of SINGULARITY, we manually

[5] Like the geometric mean, this metric is fair because if we switch SINGULARITY and SLOWFUZZ (i.e., replace $r_{b_i}^n$ with $1/r_{b_i}^n$ for all i), $WGM(\vec{r^n})$ becomes $1/WGM(\vec{r^n})$. Many other common averaging functions (e.g., arithmetic or quadratic mean) do not have this property.

Table 3: Evaluation on STAC Benchmarks.

Benchmark	Description	Input Type	DSL Used	Input budget	Target time	AV found?
blogger	Blogging web application	URL	string	5KB	300s	✓
graphAnalyzer	DOT to PNG/PS converter	DOT file	graph	5KB	3600s	✓
imageProcessor	Image classifier	PNG file	array	70KB	1080s	✓
textCrunchr	Text analyzer	text file	string	400KB	300s	✗
linearAlgebra	Matrix computation service	Matrix	array	15.25KB	230s	✓
airplan1	Online airline scheduler	Graph	graph	25KB	500s	✓
airplan2	Online airline scheduler	Graph	graph	25KB	500s	✓
airplan3	Online airline scheduler	Graph	graph	25KB	500s	✗
searchableBlog	Webpage search engine	Matrix	array	1KB	10s	✓
braidit1	Online multiplayer game	String	string	2KB	300s	✓

investigate those benchmarks for which Singularity fails to find an attack vector.

For *textCrunchr*, the root cause of the problem is the empirical measurement model. In particular, Singularity evaluates the fitness of an individual based on its performance on inputs at size 1KB, but this is much smaller than the input budget of $400KB$ and results in sub-optimal patterns. While we could circumvent this problem by using a much larger input size, that would significantly increase the time to evaluate the fitness of a given input pattern, thereby slowing down the fuzzing algorithm.

For *airplan3*, the evaluation time takes too long. During fitness evaluation, running the application on an input of size 1KB can take more than 3 minutes, and as a result, Singularity fails to converge to the fittest pattern within the 3-hour time limit.

7.5 Performance Bug Detection

To evaluate whether Singularity can help with discovering *unknown* performance bugs in real-world projects, we run Singularity on three popular Java libraries, namely Google Guava [15], Vavr [34], and JGraphT [18]. All of these libraries have more than 1000 stars on Github and are used by more than 70 other projects on Maven Central. Hence, any performance issue in these libraries is likely to have significant real-world impact.

For each library, we identify a set of public APIs related to container operations or graph algorithms and write driver code to invoke these APIs using inputs generated by Singularity. We then use the input patterns generated by Singularity to determine worst-case complexities by (a) generating inputs of different sizes, and (b) fitting a curve through these data points. If the complexity obtained by Singularity is worse than the expected worst-case, we report the anomaly to developers and let them confirm whether this is a performance bug.

Using this methodology, we identified five *previously unknown* performance bugs, all of which have been confirmed by the developers. In what follows, we include brief descriptions of the performance problems uncovered by Singularity:

Performance bugs in Guava. Singularity identified two performance bugs in the `ImmutableBiMap` and `ImmutableSet` container classes in the Guava library. Both of these classes provide a method called `copyOf` that returns an `ImmutableBiMap` or `ImmutableSet` that contains the same elements as the input collection. While both of these `copyOf` methods are expected to take linear time, the inputs generated by Singularity cause $O(n^2)$ performance. In particular, Singularity triggers this worst-case behavior by causing hash collisions despite the existence of a mechanism that tries to protect against hash collisions. The inputs generated by Singularity are complex enough to bypass these existing mitigation mechanisms. The details, including the bug report and input patterns discovered, are explained in Singularity's documentation [35].

Table 4: Graph-related Components

Signature	Description
emptyGraph()	create an empty graph
addN(g)	add a new node to the graph g
addE(g, v)	add a new edge with two new vertices and edge value v to the graph g
growE(g, v, i)	add a new edge with one endpoint being an existing node i
growLoop(g, v, i)	add a new self loop to an existing node i
bridgeE(g, v, i_1, i_2)	add an edge between two vertices $i1$, $i2$
deleteE(g, i)	delete the ith edge from graph g
mergeGraph(g_1, g_2)	merge two graphs into one graph
updateEValue(g, v, i)	update the ith edge's value in graph g
addCompleteN(g, v)	add a new node, then connect it to all existing nodes with edge value v

Performance bug in JGraphT. Singularity identified a serious performance bug in the JGraphT implementation of the push-relabel maximum flow algorithm [13]. While the theoretical worst-case behavior of this algorithm is $O(n^3)$, Singularity is able to find inputs that trigger $O(n^5)$ running time. This pattern corresponds to an RCG with 2 internal states and 3 output states, as shown below, and where the component semantics are listed in Table 4:

$$\mathcal{I} = (0, \text{addNode}(\text{emptyGraph}()))$$
$$\mathcal{F}_1 = \text{plus}(s_1, 2)$$
$$\mathcal{F}_2 = \text{growE}(\text{bridgeE}(\text{growE}(s_2, 3, 0), 4, \text{inc}(s_1), s_1), 0, 0)$$
$$\mathcal{O} = (2, 1, s_2)$$

Performance bug in Vavr. Singularity also identified two performance problems in the Vavr library that provides immutable and persistent collections. In particular, while the `addAll` and `union` methods of `LinkedHashSet` are supposed to have worst-case linear complexity, Singularity found inputs that trigger quadratic behavior. The developers have acknowledged this issue and added a caveat to the corresponding JavaDocs that these methods have quadratic rather than the (expected) linear complexity.

7.6 Threats to Validity

Randomness. Since SINGULARITY leverages randomized algorithms, its performance can be affected by various factors like parameter sampling and individual selection. Hence, our results may be skewed by unusually lucky or unlucky runs. To mitigate this concern, we run SINGULARITY (as well as SLOWFUZZ) multiple times (≥ 30) and consider the best result across all of these.

Benchmark selection. Due to their own technical limitations, we are not able to run SLOWFUZZ and WISE on a common set of benchmark programs. Instead, we compare SINGULARITY against SLOWFUZZ and WISE separately on their own benchmarks. While a common benchmark set for all tools may provide a more comprehensive view, we believe our comparison is sufficient for showing the strengths and weaknesses of these techniques.

8 LIMITATIONS

Generality. While the SINGULARITY framework can be applied to many different programs, it requires the user to provide suitable components that operate over the input type of the target program. SINGULARITY already comes with a library of components for standard data types (e.g., integers, lists, graphs), but the user needs to provide additional components for custom data types.

Driver code. While SINGULARITY supports a wide range of commonly used data types, it expects the user to write driver code to translate these DSL data structures into the format accepted by the target program. Although this kind of translation normally requires little manual effort and can even be automated, some target APIs only accept inputs with special property or of special format. In such cases, to improve fuzzing efficiency, additional effort from the user may be needed to integrate such domain knowledge into the driver code.

9 RELATED WORK

Testing for performance. There is a long line of work on automated testing techniques to uncover performance problems [4, 8, 14, 27, 32, 38, 39]. Among these prior techniques, WISE is the first one to introduce the *complexity testing* problem, where the goal is to determine the complexity of a given program by constructing test cases that exhibit worst-case behavior. At a high level, WISE uses an optimized version of dynamic symbolic execution to guide the search towards execution paths with high resource usage. While WISE is a white-box testing technique, our approach is purely black-box and can scale to larger input sizes.

From a technical perspective, PerfSyn [32] is more similar to our approach in that it uses black-box evolutionary search to generate tests that cause performance bottlenecks. Specifically, PerfSyn starts with a minimal usage example of the method under test and applies a sequence of mutations that modify the original code. However, a key difference is that PerfSyn focuses on performance bottlenecks related to API usage, whereas our approach focuses on finding input patterns that trigger worst-case complexity.

Another idea related to performance testing is *empirical computational complexity* [14]. In particular, Goldsmith et al. propose a technique for measuring empirical complexity by running the program on workloads spanning several orders of magnitude in size and fitting these observations to a model that predicts performance as a function of input size. Since this technique requires the user to manually provide representative workloads, our approach is complementary to theirs.

Performance bug detection. As argued earlier in Section 1 and demonstrated through our experiments, SINGULARITY can be useful for uncovering performance bugs. In this sense, our technique is related to a long line of work on performance bug detection [10, 23–25]. Most of these techniques target narrow classes of performance problems, such as redundant traversals [10, 23–25], loop inefficiencies [11, 22, 31], and unnecessary object creation [12]. Compared to these techniques, SINGULARITY can to detect a broader class of performance bugs but requires the user to decide whether the reported worst-case complexity corresponds to a performance bug.

Algorithmic complexity vulnerabilities. Recently, there has been significant interest in automated techniques for detecting algorithmic complexity (AC) vulnerabilities [5, 7, 9, 21, 30, 30, 37]. Some of these techniques target a specific class of vulnerabilities, such as those related to regular expressions [37]. Among approaches that target a broader class of AC vulnerabilities, SLOWFUZZ [26] is most closely related to our approach. In particular, SLOWFUZZ also uses evolutionary search for generating inputs but performs mutations at the byte level. In contrast, our method looks for input patterns rather than concrete inputs and can therefore scale better when large input sizes are required.

Asymptotic complexity analysis. Since SINGULARITY can be used to determine worst-case complexity, it is related to static techniques for analyzing the asymptotic behavior of programs [1, 3, 6, 16, 17, 29]. Our approach is complementary to static techniques in that we can generate concrete inputs that trigger worst-case behavior. For instance, our method can be used to validate the complexity bounds reported by a static analyzer and help programmers debug performance problems.

10 CONCLUSION

We have presented a new black-box fuzzing technique for generating inputs that trigger worst-case performance of a given program. The key idea underlying our method is to look for *input patterns* rather than concrete inputs and formulate the complexity testing problem in terms of optimal program synthesis. Specifically, express input patterns using recurrent computation graphs and use genetic programming to find an RCG that results in worst-case behavior. Our experiments demonstrate the advantages of our approach compared to other techniques and show that our method is useful for (a) finding worst-case asymptotic complexity bounds of interesting algorithms, (b) detecting availability vulnerabilities in non-trivial programs, and (c) discovering previously unknown performance bugs in widely used Java libraries.

11 ACKNOWLEDGEMENTS

We thank the anonymous FSE'18 reviewers, Calvin Lin, and members of the UToPiA group for their helpful feedback on earlier drafts of this paper. This work was sponsored by DARPA award FA8750-15-2-0096 and NSF Award CCF-1712067.

REFERENCES

[1] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag New York, Inc., 85–100.

[2] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 775–788.

[3] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4, Article 13 (Aug. 2016), 50 pages.

[4] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated Test Generation for Worst-case Complexity. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 463–473.

[5] Xiang Cai, Yuwei Gui, and Rob Johnson. 2009. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 27–41.

[6] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 467–478.

[7] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*. IEEE, 186–199.

[8] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 89–98.

[9] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*.

[10] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 607–622.

[11] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 895–907.

[12] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 118–128.

[13] A V Goldberg and R E Tarjan. 1986. A New Approach to the Maximum Flow Problem. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*. ACM, New York, NY, USA, 136–146.

[14] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 395–404.

[15] Google. [n. d.]. Google core libraries for Java. https://github.com/google/guava.

[16] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 127–139.

[17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 359–373.

[18] JGraphT. [n. d.]. A free Java Graph Library. http://jgrapht.org/.

[19] Alexander Klink and Julian Wälde. 2011. Efficient Denial of Service Attacks on Web Application Platforms. https://events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms.pdf. [Online; accessed 1-Feb-2018].

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, 75–.

[21] Kasper Luckow, Rody Kersten, and Corina Păsăreanu. 2017. Symbolic Complexity Analysis using Context-preserving Histories. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 58–68.

[22] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 902–912.

[23] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 562–571.

[24] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 369–378.

[25] Rohan Padhye and Koushik Sen. 2017. Travioli: A Dynamic Analysis for Detecting Data-structure Traversals. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 473–483.

[26] 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 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2155–2168.

[27] Michael Pradel, Markus Huggler, and Thomas R Gross. 2014. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 13–25.

[28] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.

[29] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *Journal of Automated Reasoning* (2017), 1–43.

[30] Randy Smith, Cristian Estan, and Somesh Jha. 2006. Backtracking Algorithmic Complexity Attacks against a NIDS. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*. 89–98.

[31] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 370–380.

[32] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 1–13.

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–.

[34] Vavr. [n. d.]. An object-functional language extension to Java 8. https://github.com/vavr-io/vavr.

[35] Jiayi Wei. [n. d.]. Singularity DSL Documentation. https://github.com/MrVPlusOne/Singularity/blob/develop/doc/GraphComponents.md.

[36] Jiayi Wei. [n. d.]. Singularity Github Repository. https://github.com/MrVPlusOne/Singularity.

[37] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 3–20.

[38] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 67–76.

[39] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. 2011. Automatic Generation of Load Tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 43–52. https://doi.org/10.1109/ASE.2011.6100093