I. Problem Definition

Definition 1. Problem configuration

A problem configuration consists of the following given elements:

- a program of interest P that takes an input of return type T (T is a tuple if P take multiple arguments)
- a resource measure function $\Re$ that takes a value of type T as input and returns the resource usage amount of running P on that value
- a size metric function *size* that maps an input value into an integer
- a size of interest $\hat{s}$ and a resource usage bound $\hat{r}$

Now we can define algorithmic complexity vulnerabilities as follows:

Definition 2. Algorithmic complexity vulnerability

We say there is an algorithmic complexity vulnerability (ACV) iff there exists an input x whose size is smaller than $\hat{s}$ and whose resource usage exceeds the given bound $\hat{r}$. i.e.

$$\text{Vulnerable} \equiv \exists x.(size(x) \leq \hat{s} \wedge \Re(x) \geq \hat{r})$$

Problem 3. ACV attacking problem

Assuming there is an ACV, we want to find a proof input x such that $size(x) \leq \hat{s} \wedge \Re(x) \geq \hat{r}$.

For every size s, there exists a *max case input* $\hat{x}$ s.t. $size(\hat{x}) \leq s$ and $\forall x.(size(x) \leq s \longrightarrow \Re(\hat{x}) \geq \Re(x)$. If we plot all max cases inputs on a resource usage chart (size as x axis and resource usage as y axis) and connect them, we get a max resource curve.

Definition 4. Max resource curve

The max resource curve $\mathit{\Gamma}$ is the following function

$$\mathit{\Gamma}(s) \equiv \max_{x \,|\, size(x) \leq s} \Re(x)$$

Note that a max resource curve is by definition monotonously increasing.

In figure I.1, line $\mathit{\Gamma}$ represents a max resource curve, and point P is the resource bound point. It is easy to see that there is an algorithmic complexity vulnerability iff P is below $\mathit{\Gamma}$. The red region is called the vulnerable region, and we can solve the ACV attacking problem by constructing any input that lies within it.

In this paper, we focus on those problem configurations where the size of interest $\hat{s}$ is large and some common pattern is shared among many max case inputs.

These kinds of problems are important because many complexity vulnerabilities [...]

These kinds of problems are also challenging for traditional fuzzing techniques because as $\hat{s}$ grows, the potential input space grows exponentially and creates an infeasibly large search space. On the other hand, as we shall see in the following examples, many real world ACVs can be triggered by utilizing some special input patterns; hence, it is often easy to construct sufficiently large malicious inputs once we have found those patterns.

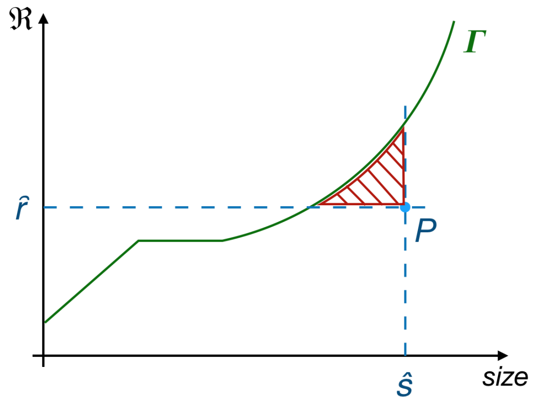

Figure I.1. Resource usage chart
Green line $\mathit{\Gamma}$ represents a max resource curve. The blue point P is the resource bound point. The red region is called the vulnerable region, and we can solve the ACV attacking problem by constructing an input that lies within it.

Definition 5. Input sequence

An input sequence X is an *infinite* sequence of values $X_0, X_1, X_2, ...$ that satisfy the requirement $\forall i.(size(X_i) < size(X_{i+1}))$

For example, if we choose T to be vector of integers and $size$ to be vector length, then both $[1]$, $[1,2]$, $[1,2,3]$, $[1,2,3,4], ...$ and $[]$, $[0,0]$, $[0,0,0,0]$, $[0,0,0,0,0,0], ...$ are input sequences, but $[1]$, $[2]$, $[3]$, $[4], ...$ is not.

Example 6. Quicksort with middle pivots

Let P be a quicksort program that takes a vector as input, also assume it always selects the middle $(length/2)$ element in its input as pivot. $size$ is chosen to be vector length, and $\Re$ is how much time it takes (or lines of code executed) to run P. To exploit quicksort's $O(size^2)$ worst case behavior, we are interested in finding a max case input for some large $\hat{s}$.

In order to trigger this worst case behavior, during each pivot selection, either the smallest or the largest element in the input vector should be selected, and all other elements would then get sent into the next recursive call. One way to construct such an input is by starting with an empty vector and repeatedly appending and prepending larger elements to it, resulting the input sequence $[\,], [0], [1,0], [1,0,2], [3,1,0,2]...$ Actually, it is easy to see that this input pattern forms a max resource curve; thus, we can easily get a max case input for $\hat{s}$ after finding this sequence.

-

Example 7. (Another example here?)

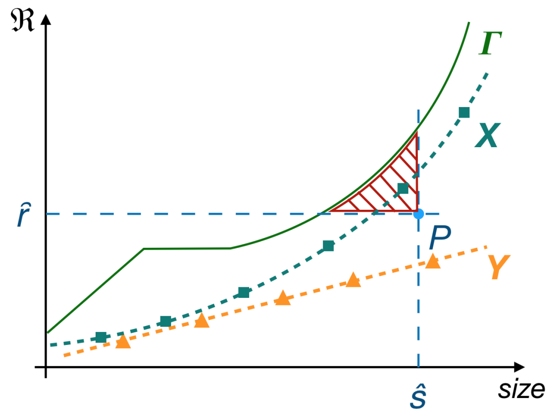

Figure II.1. Effective and ineffective patterns
In this chart, pattern X is an effective pattern because it contains one point that lies within the vulnerable region. On the other hand, pattern Y can not trigger an ACV and is thus not an effective pattern.

II. Methodology

A. An Optimization Problem

Based on the above observations, we propose a new approach for solving ACV problems. The key idea is that, instead of directly searching for large inputs, we search for input patterns that can generate large inputs. Using the resource usage chart definition, the problem now becomes searching for an input pattern whose points intersect with the vulnerable region, and we call such patterns an effective pattern. see Figure II.1.

In general, the number of possible patterns can be huge or even infinite for a given problem configuration, and it is impractical to enumerate all of them. To efficiently search for correct patterns, we made the following two restrictions/assumptions to simply our problem:

1) We introduced a component-based domain specific language (DSL) called *recurrent computation graphs*. RCGs can be used to generate different input patterns, and we only search for patterns expressible by this language.
2) We assume program P's resource usage behavior has some continuity in the sense that similar input patterns would have similar resource usages, and as a result, we can efficiently guide the searching process by treating P's resource usage as a useful feedback.

Under these settings, we can turn problem 3 into the following equivalent optimization problem:

Definition 8. Pattern performance
A pattern X's performance under size s is defined to be the maximal resource usage of X's elements whose size does not exceed s.

$$\Re(X, s) \equiv \max_{x \in X \wedge size(x) \leq s} \Re(x)$$

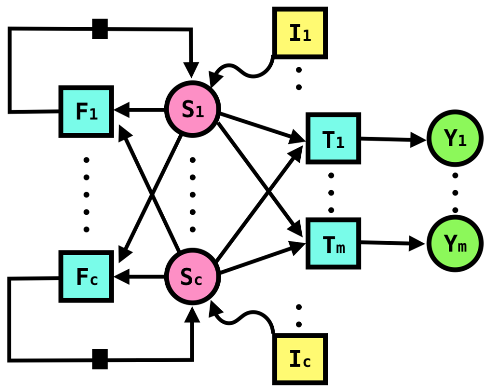

Figure II.2. Recurrent computation graph

Note that in general, $\Re(x)$ may not be monotonously increasing, that is why we need to take the max in the above definition.

Problem 9. Input pattern optimization
Given a DSL $\mathscr{D}$, let $\mathbb{P}$ be the set of all patterns expressible in $\mathscr{D}$, we want to find an input pattern with the maximal performance under our size of interest $\hat{s}$, i.e. find

$$\underset{X \in \mathbb{P}}{argmax}(\Re(X, \hat{s}))$$

Having the optimization goal set up, next, we introduce the DSL we use in our algorithm.

B. Recurrent computation graph (RCG)

A recurrent computation graph of shape (c, m) is a triple (I, F, T), in which I is a tuple of c initialization values, F is a tuple of c state updating functions, and T is a tuple of m output functions. Here c is called the model capacity, and it determines how many internal states can be used during pattern generation. m equals to the number of arguments our program P takes (hence, m is the length of tuple T). We can then generate an input pattern Y as follows: First, we set up c internal states s_1 through s_c and initialize them using I_1 through I_c, respectively. Then, we construct the first output $(y_1, y_2, .., y_m)$ by evaluating T_1 through T_m using s_1 through s_c as input arguments. After that, during each time step, we use F to update all internal states, which are again used by T to evaluate the next output. This process is demonstrated by Figure II.2

To represent I, F, and T, we use a set of first-order, generically-typed expressions defined by the following grammar:

```
const := value
       | f(const_1, ..., const_a)
func  := x_i  (i ∈ [1...c])
       | const
       | f(func_1, ..., func_a)
```

where $const$ is I's grammar, and $func$ is F and T's grammar. f belongs to some extendable component set $\mathcal{F}$, a is the arity of f, and x_i's are function arguments.

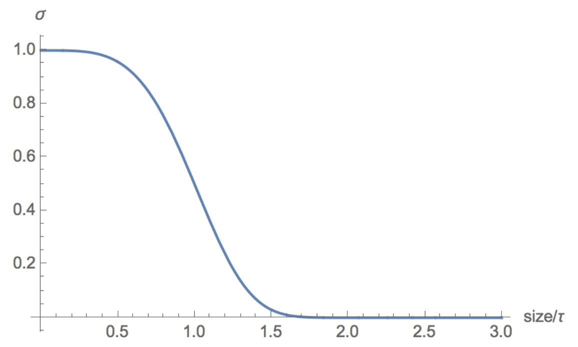

Figure II.3. Size penalty function σ
In our implementation, σ is chosen to be $\exp(-(size/\tau)^4)$. When $size < 0.5$, $\sigma \approx 1$; at $size = \tau$, $\sigma = 0.5$; when $size > 1.5\tau$,$\sigma \approx 0$

C. *Feedback guided optimization*

Because our goal here is to perform optimization over a discrete domain of computation graphs, we use an Evolutionary Optimization Algorithm in our implementation. [Why are Evolutionary Algorithms good & Some introduction to Genetic Programming]

We developed a new Genetic Programming algorithm which works for recurrent computation graphs and is guaranteed to produce input patterns with correct types. Below, we describe the details of this GP algorithm.

1) Representation: The genotype representation of an individual is just an RCG; our GP algorithm directly modifies the ASTs of computation graphs' expressions.

Our phenotype representation is input patterns. We can transcribe a genotype into a corresponding phenotype by evaluating the RCG using the method previously described in Figure II.2.

2) Fitness function: Fitness function is used to assess the performance of each individual in achieving the optimization goal. Individuals with higher fitness are more likely to be selected to participate in the creation of the next generation. For an RCG g, denoting its corresponding pattern as X^g, then our fitness function is chosen to be the performance of X^g times some size penalty factor $\sigma(size(g))$. Here we define the size of an RCG to be the total number of AST nodes used.

We choose this form of fitness for two reasons:

Bloat control. One of the fundamental problems of GP is bloat. After some generations of evolution, the search for better programs halts as the programs become too large. Multiplying a size penalty can reduce evolution's tendency towards larger programs. We choose σ to be some function of a "cliff" shape: when the size is small, σ is very close to 1, but as the size exceeds some threshold τ, σ quickly drops to nearly 0. By using this penalty function, individuals' sizes are very unlikely to exceeds this threshold, and when performance is the same, smaller individuals are always preferred. σ's shape is shown in Figure II.3.

Pattern simplification. We also found this size penalty factor act as a form of Occam's razor that helps both the

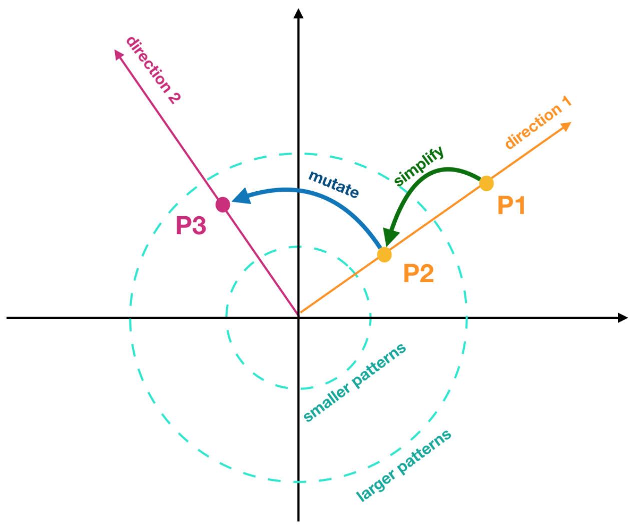

Figure II.4. Simplification helps evolution
Above is a 2D projection of $\mathbb{P}$, the space of all possible patterns. P1 is a local optimum, P3 is a better pattern but is very far away from P1. Thus, any random mutation is unlikely to change P1 into P3. By simplifying P1 into P2 first, although the performance does not improve, since it is easier to mutate P2 into P3, the probability to find P3 increases largely.

performance of GP and increases the generality of the found patterns. In the results of our experiments, it is very common to see this pattern: after GP reaches some local optimum, the best pattern's performance stops to improve because any deviation from this local optimum tend to decrease the performance; nevertheless, the optimizer keeps simplify those patterns to make their fitness closer to their actual performance. After many individuals get simplified, some are brought by random mutations into previously unseen, better evolution directions. Since then, the previous local optimum is escaped and further performance improvements follows in the next several generations. This process is demonstrated in Figure II.4

3) Genetic operators: In GP, genetic operators are used to create new individuals from old ones. In our implementation, we have developed three kinds of genetic operations, namely, mutation, crossover and copy. These are the most common operators in GP, but our system is fully extendable and allows users to create new operators when needed. When creating a new individual, each operator φ have a probability $P(\varphi)$ of being used as the reproduction method (P is an adjustable hyper parameter), and depending on the arity of φ, a number the participates are then chosen among the old population by a random selection method called the tournament method.

Mutation. Mutation randomly modifies an RCG to produce a new RCG. It is implemented as follows: First, it randomly chooses an AST node from all expressions used by the RCG (i.e., from I,F, and T) . The chosen AST node is called the mutation point. Then, it randomly generates a new AST of the same return type and uses it to replace that mutation point. Note that if the mutation point is from the initialization values I, mutation will replace it with a constant expression; in other cases, the newly generated expressions can either be a constant or a function.

Crossover. Crossover creates a new individual by mixing two old ones. It randomly chooses two mutation points of the same return type from both participated RCGs and swaps them. This results in two new individuals, and either one can be selected as the result of this operator.

Copy. Operator copy just brings an individual to the next generation without any modification. It is used to provide some stability to the GP process. Because individuals with better fitness measure are more likely to be chosen by the tournament method, a small $P(Copy)$ is usually sufficient to guarantee that the best individual will be copied into the next generation.