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 type T (T is a tuple if P take multiple arguments)
- a size metric function *size* that maps a value of type T to an integer
- 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 of interest $\hat{s}$ and a resource usage bound $\hat{r}$
- We also assume that it is always possible to increase the resource usage by constructing bigger inputs, i.e. $\forall x.\exists x'.size(x) < size(x') \wedge \Re(x) < \Re(x')$

Now we can define algorithmic complexity vulnerabilities as follows:

Definition 2. Algorithmic complexity vulnerability (ACV)

We say there is an algorithmic complexity vulnerability 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 a 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 $\forall x.(size(x) \leq s \longrightarrow \Re(x) \leq \Re(\hat{x})$. 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 Γ is the following (monotonously increasing) function

$$\Gamma(s) \equiv \max_{x \mid size(x) \leq s} \Re(x)$$

In figure I.1, the green line Γ represents a max resource curve, and the blue cross P is the resource bound point. It is easy to see that there is an algorithmic complexity vulnerability iff P is below Γ. And we can solve the ACV attacking problem by constructing any input that lies within the red region (called vulnerable region).

In this paper, we focus on those problem configurations where the size of interest $\hat{s}$ is large and there exists some inner pattern within max case inputs. This kind of problems are challenging for traditional fuzzing techniques because as $\hat{s}$ grows, the potential input space grows exponentially and result in 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 inputs of some special patterns, hence it is often possible to construct sufficiently large malicious inputs by utilizing those patterns.

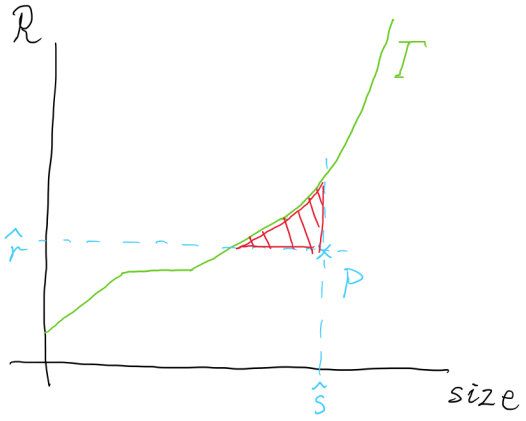

Figure I.1. Resource usage chart

Definition 5. Input pattern

An input pattern 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 $[0], [0,0,0], [0,0,0,0,0], ...$ are input patterns, 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, and choose $size$ to be vector length. $\Re$ is how much time it takes (or lines of code executed) to run P on some input. We are interested in finding a max case input for some large $\hat{s}$.

Also assumes that this quicksort implementation always selects the middle $(length/2)$ element in its input as pivot. Then $[7,5,3,1,0,2,4,6]$ is a max case input for size 8. There is a clear pattern in this input, and similarly, we can construct a series of max case inputs by starting with an empty vector and repeatedly append and prepend larger elements to it, and this results in $[], [0], [1,0], [1,0,2], [3,1,0,2]...$ 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 pattern.

-

Example 7. (Another example here?)

II. Methodology

A. An Optimization Problem

According to above observations, we propose a new approach for solving ACV problems. The key idea is that, instead of directly searching for a large input, we search for input patterns. Using a resource usage chart as demonstration, the problem now becomes searching for an input pattern whose points intersect with the vulnerable region,

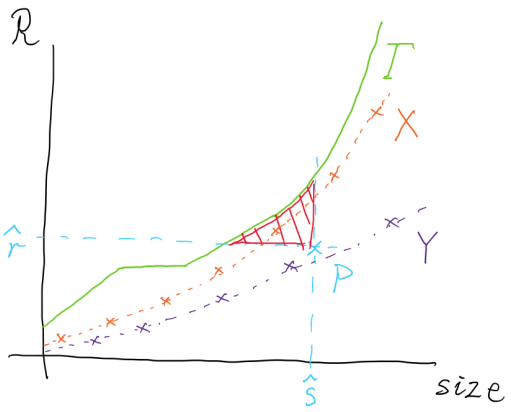

Figure II.1. Effective and ineffective patterns
In this chart, pattern X is an effective pattern because it contains one point that lies within the vulnerable region. On the other hand, pattern Y can not trigger an ACV and is thus not an effective pattern.

and we call such patterns an effective pattern. see Figure II.1.

But 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* which can be used to generate different input patterns, and 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:

Problem 8. 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 such that it has the maximal resource usage before exceeding $\hat{s}$, i.e. find

$$\underset{X\in\mathbb{P}}{argmax}(\max_{x\in X_{\hat{s}}}\Re(x))$$

where $X_{\hat{s}} \equiv \{x|x \in X \wedge size(x) \leq \hat{s}\}$

Now we introduce the DSL we are using in our algorithm:

B. Recurrent computation graph (RCG)

A recurrent computation graph of shape (c, m) is a triple (I, F, T), where 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 model capacity, it determines how many internal states can be used during pattern generation, and m is the number of arguments program P takes (hence, m is the length of tuple type 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 can construct the first output $(y_1, y_2, .., y_m)$ by calculating T_1 through T_m using s_1 through s_c input arguments. After that, during each time step, we use F to update all internal states, which are then used by T to calculate new outputs. This process is demonstrated by Figure II.2

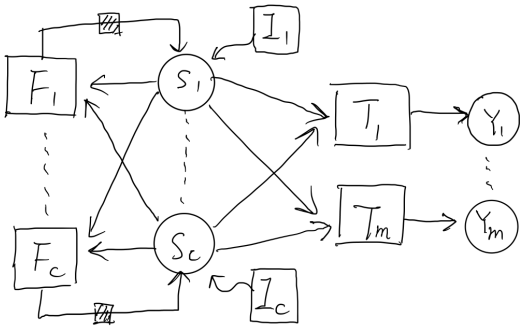

Figure II.2. Recurrent computation graph

To represent I,F, and T, we use first-oder, generically typed expressions with the following grammar:

$$\begin{aligned}
&\texttt{const} := \texttt{value} \\
&\qquad \mid \texttt{f}(value_1, ..., value_a) \quad (f \in \mathcal{F}) \\
&\texttt{expr} := x_i \quad (i \in [1...c]) \\
&\qquad \mid \texttt{const} \\
&\qquad \mid \texttt{f}(expr_1, ..., expr_a) \quad (f \in \mathcal{F})
\end{aligned}$$

where $\mathcal{F}$ is some extendable component set.

C. Feedback guided optimization