University of Colorado, Boulder

Master Level Independent Study - Report

CSCI 5900-935

Generating Structured String Inputs for CPU Exhaustion Attacks

Author:
Santhanakrishnan Ramani

Supervisor:
Dr. Pavol Cerny

May 4, 2017

University of Colorado Boulder

Abstract

In this study, we propose a technique for generating CPU exhaustion attack vectors – structured string inputs (i.e., valid XML or DOT files) using context free grammar and evolutionary techniques that can cause the server to run for extended periods of time or consume more memory than a certain threshold (exponential in the size of the program input) when passed as an input to the respective application. The technique performed really well on some of the benchmark examples used to measure it.

1 Introduction

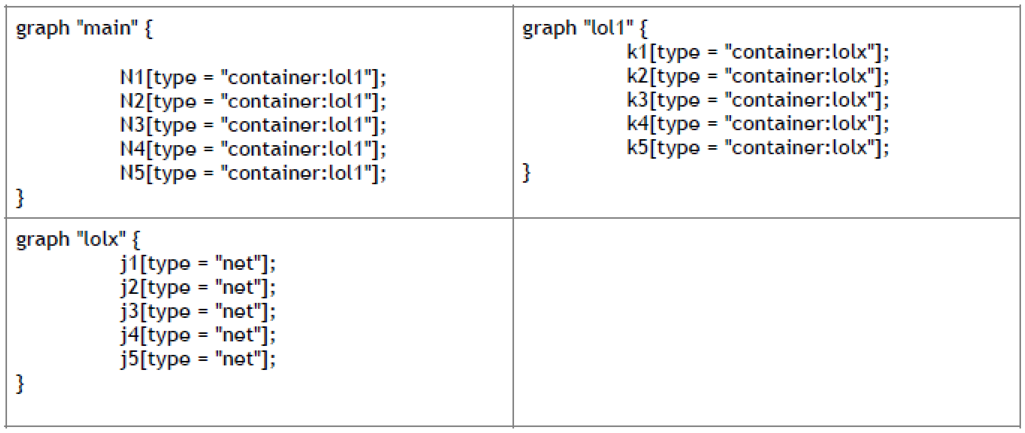
This idea or technique of generating structured string inputs automatically for a given application was started and was put into place when we were solving DARPA engagement problems as a part of the STAC project undertaken by the CUPLV (CU Programming Language and Verifications Lab). The main goal here is to come up with a well defined and crafted valid input for the particular application that causes a Denial-of-service attack under the given input budget.

In order to come up with this sort of crafted input initially we came up with a technique that takes parameterized regular expression that would generate the required type of input which is then passed to some well known evolutionary algorithms like CMAES (Covariance Matrix Adaptation Evolution Strategy), SDS (Stochastic Diffusion Search) whose value function is modified accordingly depending on whether memory, time or count of particular loop of the application needs to be measured. This technique worked quite well for smaller applications or applications which took in simpler inputs. Later when we had to find a crafted input for an application that needed a DOT (graph description language) as an input which had a more formal structure we thought of coming up with a more generic approach that could tackle any problems of this kind, that's when the idea of passing a context free grammar as input struck us. There were many challenges on road to take the given grammar and come up with the attack vector that can be formed by following the rules of grammar inputted. In this technique, we make use of tools like ANTLR a parser generator, global optimization techniques like CMAES, SDS and Mosek an ILP (Integer Linear Programming) solver to handle the challenges.

2 Motivating Examples

2.1 Dot Parser Example

The challenge was to generate a DOT input that causes space utilization attack on a real world application a GUI based graph viewing tool. The tool is capable of visualizing arbitrary DOT graph loaded from disk, and exporting them to vector and raster based image formats. To use the tool, the user first selects an input which is a file with a graph described using some given specification. Then the user can choose to export the graph where it is fully expanded and visually laid out. The graph layout can be done either using random layouts or force-directed layouts. These laid out graphs can then be rendered to either PNG or Postscript format. Since Postscript is a vector format the expanded graph must have separate drawing instructions for each node in the graph, the output file can be extremely large relative to the input. What we didn't know beforehand was the DOT parser was intentionally modified to create a keyword called "container", which allows for a node in the graph to be represented by an arbitrary DOT graph and the container objects are allowed to have nested containers. A sample vulnerable input to the tool is shown in figure 2.1.



```
graph "main" {

            N1[type = "container:lol1"];
            N2[type = "container:lol1"];
            N3[type = "container:lol1"];
            N4[type = "container:lol1"];
            N5[type = "container:lol1"];
}
```

```
graph "lol1" {
            k1[type = "container:lolx"];
            k2[type = "container:lolx"];
            k3[type = "container:lolx"];
            k4[type = "container:lolx"];
            k5[type = "container:lolx"];
}
```

```
graph "lolx" {
            j1[type = "net"];
            j2[type = "net"];
            j3[type = "net"];
            j4[type = "net"];
            j5[type = "net"];
}
```

Figure 1: Billion graphs example attack

This graph attack here is fundamentally equivalent to the well known billion laughs space utilization XML attack.

2.2 Billion laughs attack

A billion laughs attack is a type of denial-of-service attack that targets XML parsers. It is also commonly referred to as an XML bomb or as an exponential entity expansion attack. The beauty of this attack is that it can achieve using well-formed XML thereby passing XML schema validation, which makes it hard to detect and figuring out ways to mitigate the threats caused by it. In an XML, an entity is a symbolic representation of information, just like a variable in a computer program and it must be declared in the Document Type Definition (DTD), just like an element or an attribute. To use the entity defined in the XML document content section, you need to use an ampersand (&) followed by the entity name you specified in the DTD followed by a semicolon (;). The code below illustrates an example of billion laughs attack. It is achieved by defining 10 entities, each defined as consisting of 10 of the previous entity, with the document consisting of a single instance of the largest entity, which expands to one billion copies of the first entity.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">

                                   ⋮

 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Figure 2: XML "Billion Laughs" example caption...

When the above example is passed to an XML parser, the parser sees that it contains only one root element "lolz", that contains the text "& lol9;". But, "& lol9;" is defined as an entity that expands to ten "& lol8;" strings. Which in turn is a defined entity that expands to ten "& lol7;" strings and so on. Once all the entities have been expanded this small ($< 1KB$) block of XML will actually contain 10^9 = a billion "lol"s which takes about 3 GBs of memory. There are other variants of the above-described attack too.

3 Overview

3.1 Context Free Grammar

In formal language theory, a context-free grammar (CFG) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. In CFG, all rules are one-to-one, one-to-many, or one-to-none. These rules can be applied regardless of context. The left-hand side of the production rule is also always a nonterminal symbol. This means that the symbol does not appear in the resulting formal language. Let's see a simple example of CFG that describes all two letter string containing the letters α and β. The production rule is described below,

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow \alpha \\ A &\rightarrow \beta \end{aligned}$$

If we start with the nonterminal symbol S then we can use the rule $S \rightarrow AA$ to turn S into AA. We can then apply one of the two later rules. For example, if we apply $A \rightarrow \beta$ to the first A we get βA. If we then apply $A \rightarrow \alpha$ to the second A we get $\beta\alpha$. Since both α and β are terminal symbols, and in CFG terminal symbols never appear on the left hand side of a production rule, there are no more rules that can be applied.

3.1.1 Parameterized CFG

It is same as the CFG but with an restriction on how many times a particular rule can be applied. The rules are restricted so as to generate inputs within the given input budget. For eg, let's consider the production rule given below,

$$\begin{aligned} S &\rightarrow A\_\{1\} \,|\, \alpha \\ A &\rightarrow S\_\{1\} \,|\, \beta \end{aligned}$$

If we start with the nonterminal symbol S and say we apply the rule $S \rightarrow A$ to turn S into A. We can then apply either $A \rightarrow S$ or $A \rightarrow \beta$. Say, we apply $A \rightarrow S$ we won't be able to chose $S \rightarrow A$ again as it can only be used at most once and we have to apply $S \rightarrow \alpha$ instead and there will be no more rules that can be applied.

3.2 ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks.

From a formal language description called a grammar, ANTLR generates a parser for that language that can automatically build parse trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code.

For the purpose of this project, we created our own ".g4" file which defines the rules that the inputted grammar needs to follow, the variables that can be used to represent terminals, non-terminals and how to define the cost of applying a particular rule, etc. These rules and definitions will then later be used by the ANTLR IDE to validate the grammar inputted to this tool before it automatically converts the given grammar to code and tokens.

3.3 Mosek

It is a industrial grade tool for solving mathematical optimization problems such as:

- linear programs
- quadratic programs
- conic problems
- mixed integer problems.

Here we made use of the Java interface of MOSEK to solve two complex ILP problems derived from the inputted parameterized grammar. First one gives us the frequencies of each production rule on the right hand side and the second gives us the order in which the production rules must be applied to get the intended input using the solution from first ILP. We will see how it is achieved with a detailed example in the framework section below.

Note: While using MOSEK the matrices that's being passed to the solver should be passed in column-major order.

3.4 Global Optimization Algorithms

3.4.1 CMAES

It is one of the most powerful evolutionary algorithms for single-objective real-valued optimization.

CMA-ES algorithm accepts the number of dimensions n as input and starts with an initial distribution with mean μ_0 and covariance matrix C_0. CMA-ES samples m points (where $m > 0$) according to the initial distribution μ_0, C_0. Consider that $\vec{x}_1, ..., \vec{x}_m$ be the sampled points ranked in the decreasing order based on their fitness values $f(\vec{x}_1), ..., f(\vec{x}_m)$, computed by the FC component. At each evolution step i, CMA-ES updates the distribution μ_{i+1}, C_{i+1} based on the sampled points $\vec{x}_1, ..., \vec{x}_m$. In general, these points are weighted in importance so that the resulting distribution has the highest density around the points with the highest fitness values and the lowest density around the points with the lowest fitness values. In practice, we compute a fixed number of evolutionary attempts, given by a user-defined threshold, and choose the input with the highest fitness value as a near optimal input [1].

3.4.2 Stochastic Diffusion Search

SDS came into existence in 1989 as a population-based, pattern-matching algorithm [2]. Unlike stigmergic communication employed in Ant Colony Optimization, which is based on modification of the physical properties of a simulated environment, SDS uses a form of direct (one-to-one) communication between the agents similar to the tandem calling mechanism employed by one species of ants, *Leptothorax Acervorum* [3].

It consists of four phases namely initialize, test, diffusion, and convergence. In the initialization phase, each agent is assigned a possible hypothesis and stochastically employed in the search space. In the test phase, all agents evaluate the partial objective function according to their current hypothesis and the result of this evaluation is binary making them active or inactive. In the diffusion phase, the inactive agent's, A randomly picks another agent, B if he is active, A copies B's hypothesis, else randomly selects another hypothesis over the entire search space. The test and diffusion phase keep repeating until the termination criteria are reached.

4 General Framework

Our framework basically works as described in the flowchart below,

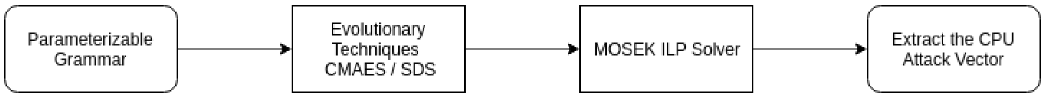

Parameterizable Grammar
Evolutionary Techniques CMAES / SDS
MOSEK ILP Solver
Extract the CPU Attack Vector

Figure 3: Framework Flowchart

The pseudo code below gives us a overall picture of how the tool works in general. We will see an detailed explanation of how each step with an example in the upcoming sections.

Pseudo code

```
Data: Parameterizable Context Free Grammar
Result: Attack vector String
set target_fitness_value;
set max iterations;
while not reached max iterations do
    params = sample(Evolutionary Technique);
    CFG = generate_PCFG(params);
    frequency = Grammatical ILP solver(CFG);
    solution = Grammatical ILP solver(frequency, CFG);
    string = extract_String(solution);
    if fitness_value(string) > target_fitness_value then
        print result;
        break;
    end
end
```

Algorithm 1: Algorithm to generate Attack Vectors

4.1 Sample Parameterized Grammar

Lets consider a sample parameterized grammar that looks like the one below,

$$
\begin{aligned}
S &\to (aA)\_\{\} \,|\, (bB)\_\{\} \\
A &\to (aCb)\_\{\} \,|\, a \\
C &\to (aB)\_\{\} \,|\, (A)\_\{\} \,|\, c \\
B &\to (cA)\_\{\} \,|\, (C)\_\{\} \,|\, b
\end{aligned}
$$

start S

cost a 1, b 2, c 3

replace a "1", b "01", c "11"

The above grammar will be fed as an input to one of the evolutionary techniques defined above and the "{}" will be replaced by integer values based on what the algorithm chooses at every iteration. Lets say the above grammar gets transformed to grammar shown below,

$$
\begin{aligned}
S &\to (aA)\_1 \,|\, (bB)\_1 \\
A &\to (aCb)\_2 \,|\, a \\
C &\to (aB)\_2 \,|\, (A)\_3 \,|\, c \\
B &\to (cA)\_3 \,|\, (C)\_3 \,|\, b
\end{aligned}
$$

start S

cost a 1, b 2, c 3

replace a "1", b "01", c "11"

4.1.1 Cost & Replace in CFG

We define a cost for each terminal used in our grammar depending on the particular function it denotes. This cost of terminals plays an important role in the solution we obtain out of the ILP as it is an integral part while constructing the objective function of the ILP which we will see in detail in the following sections. And, replace function helps us to define a substitute value for all terminals which gives us back the input needed to be feed to the application for which we are trying to come up with an attack vector.

4.2 Converting grammar to ILP

In order to convert the grammar to ILP, we assign variables to denotes the frequency of all the rules on the right side of the production rule.For eg., aA is denoted as f_1, bB is denoted as f_2 and so on. And W() denotes the cost of choosing that particular terminal variable.The above grammar when converted to ILP looks like this,

Objective Function:

$$\max w(a) * (f_1 + f_3 + f_4 + f_5) + w(b) * (f_2 + f_3 + f_10) + w(c) * (f_7 + f_8)$$

Constraints:

$f_1 + f_2 = 1$ // in(S) = out(S)

$f_1 + f_6 + f_8 - f_3 - f_4 = 0$ // in(A) = out(A)

$-f_2 - f_5 + f_8 + f_9 + f_10 = 0$ // in(B) = out(B)

$-f_3 - f_9 + f_5 + f_6 + f_7 = 0$ // in(C) = out(C)

The constraints are defined based on a simple law of conservation. The frequencies on the left side of the production rule must be equal to frequencies on the right side of the production rule.

On solving the above ILP using MOSEK, we get the following values for frequencies $[f_1, f_2, .., f_{10}]$ - $[0, 1, 2, 1, 2, 0, 0, 3, 0, 0]$ and an optimal cost of 20.

4.3 Retrieving back the String from ILP

Once we get back the frequency of each rules from the ILP solver the problem isn't solved yet as we still don't know the sequence in which these rules got applied. In order to find it out, we pass this solution to the next ILP solver to get back the order which will further be used to extract the solution out from the grammar.

A detailed explanation of how it is achieved and how the ILP's objective function and constraints are formed is provided in the section below.

Setup

As we know the frequencies of each rule the sum of it gives the total production rules (say F) that needs to be applied to get the terminal String. Here, x_vars represent state variables that correspond to non-terminals present in the rule that we can expand is duplicated F+1 times (i.e., initial state, F transitions resulting in the final state) and r_vars represents production rule variables a boolean which states either a production rule is used or not at a given step, it is duplicated F times once for each transition.

Our goal is to find the order in which rules are applied so as to form a valid production i.e., we start in state (1, 0, ... 0) [only start symbol nonterminal present] and we want to reach state (0, 0, ... 0) [i.e., terminal string] in exactly F transitions subjects to the following constraints. [4]

4.3.1 Constraints

- "Only one rule at each step" (i.e. we want at least one rule to be used at a transition step and, similarly only at most one rule can be used on any transition step).
- "Frequency constraints" (i.e., we know from first ILP how many times in total each production rule can be used so the $\sum_{0 \leq i < F} r_{ij} = f_j$, for production rule j).
- "Effects on state" (available nonterminals) of taking a specific rule on each step. (i.e. If I take S $\rightarrow$ AB, then that decrements my S count and increments A, B counts in the state)
- "Rule applicability (no bogus rules)" (i.e., rules can only be applied if the nonterminal count on the LHS of a production rule is non-zero in the current state. As an example, if I am in the state "SaA" I am not allowed to use a rule B $\rightarrow$ bb.

4.3.2 Solution

Finally we extract solution simply by walking through the order captured in the $r_{00}, ..., r_{0P}, r_{10}, ..., r_{FP}$ (where F is #transitions, P is #production rules in grammar).

Example

If we run the second ILP for the grammar we have been considering so far by following the steps stated above we get the order in which production rule gets applied at every step as shown in the table below.

Step	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
1	0	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	0	0
3	0	0	1	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0
6	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0
9	0	0	0	1	0	0	0	0	0	0

Table 1: Order in which rule gets expanded

The steps below shows us how the intended string is extracted using the grammar production rules and the solution obtained in the previous step.

$$\begin{aligned}
S &\rightarrow b\mathbf{B} && f_2\ rule\\
&\rightarrow bc\mathbf{A} && f_8\ rule\\
&\rightarrow bca\mathbf{C}b && f_3\ rule\\
&\rightarrow bcaa\mathbf{B}b && f_5\ rule\\
&\rightarrow bcaac\mathbf{A}b && f_8\ rule\\
&\rightarrow bcaaca\mathbf{C}bb && f_3\ rule\\
&\rightarrow bcaacaa\mathbf{B}bb && f_5\ rule\\
&\rightarrow bcaacaac\mathbf{A}bb && f_8\ rule\\
&\rightarrow bcaacaacabb && f_4\ rule
\end{aligned}$$

At last the string gets replaced by the actual values to get the intended string which in this case is *"**01111111111110101**"*.

5 Experiment Setup

Below mentioned are the steps followed to setup the experiment. For a more detailed explanation with images kindly take a look at the supplementary section at the end.

Step 1: If you don't have eclipse installed on your PC or laptop kindly download it from here and install it.

Step 2: Next you need to get the MOSEK license from here in case you don't have one already.

Step 3: Clone the cpu-attacks source code from here.

Step 4: Open Eclipse, Click File → Open Projects from File System. Then in the pop up add the path of the Directory cpu-attacks/src in the Import Source text area and hit Finish.

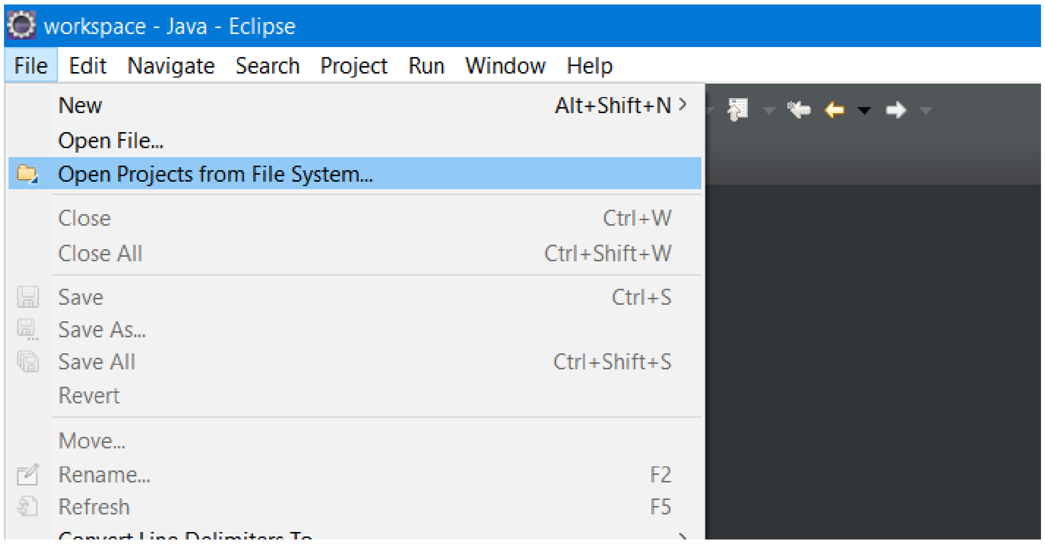
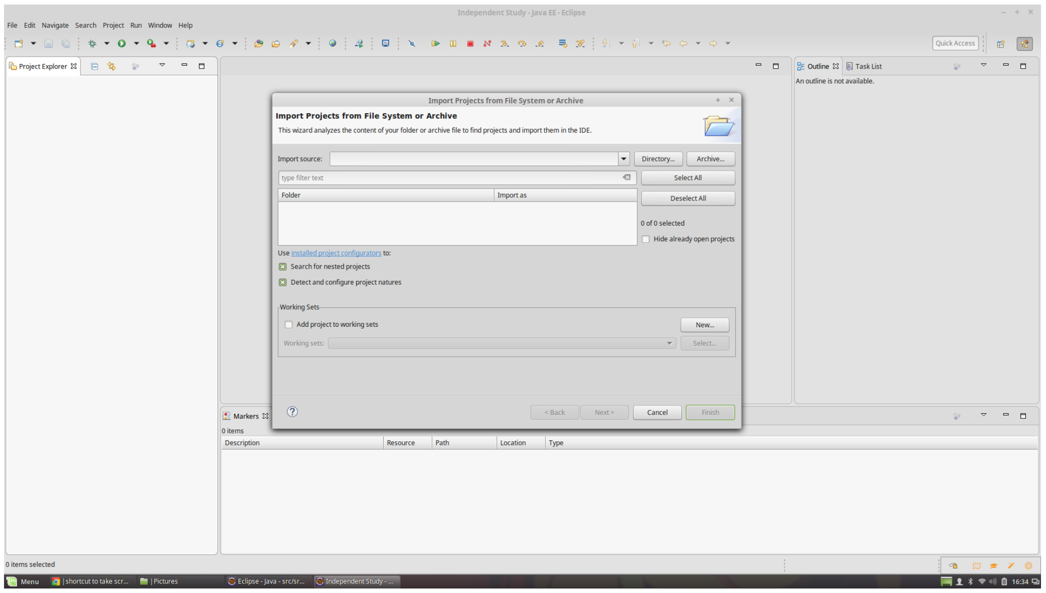


Figure 4: Open Project



Import Projects from File System or Archive

This wizard analyzes the content of your folder or archive file to find projects and import them in the IDE.

Import source:

type filter text

Folder	Import as

0 of 0 selected

Hide already open projects

Use installed project configurators to:

Search for nested projects

Detect and configure project natures

Working Sets

Add project to working sets

Working sets:

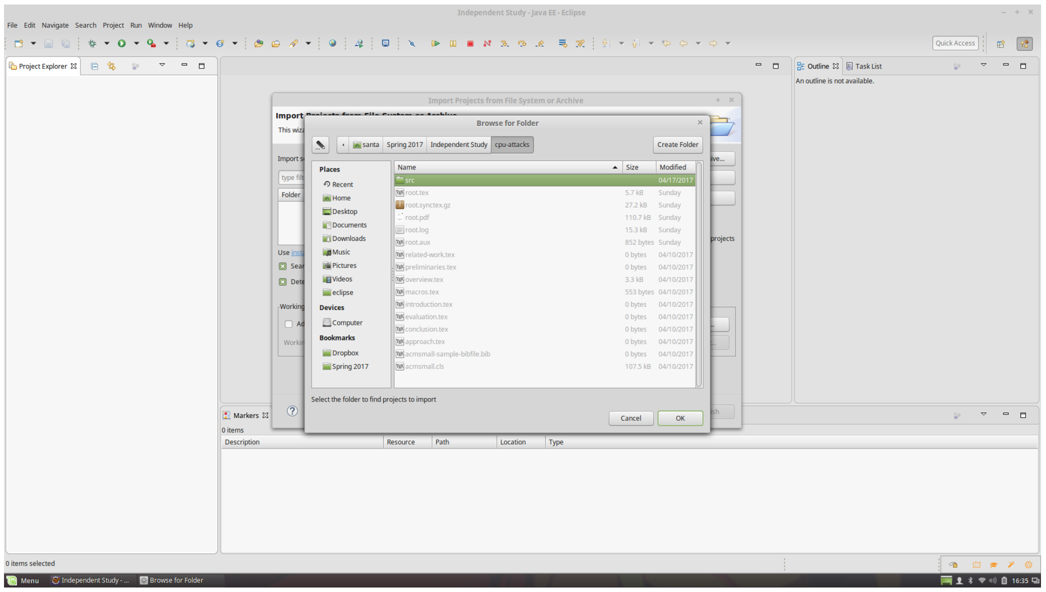
An outline is not available.

0 items

Description	Resource	Path	Location	Type

0 items selected

Figure 5: Import Projects From File System

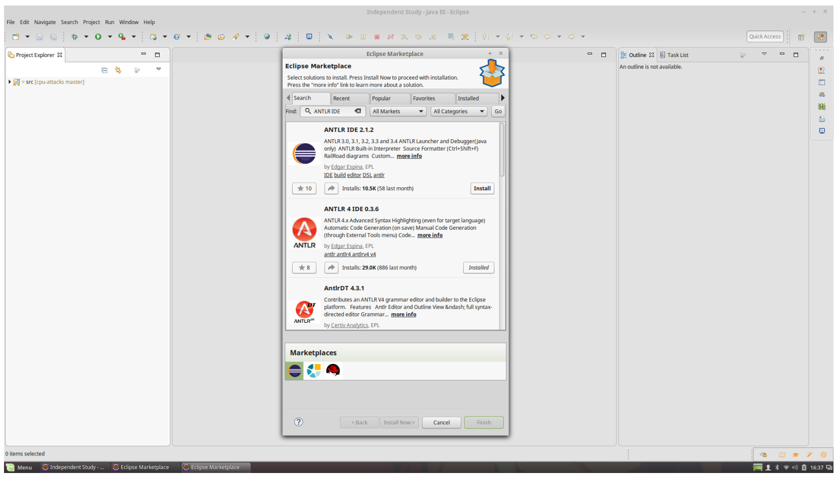


Name	Size	Modified
src		04/17/2017
root.tex	5.7 kB	Sunday
root.synctex.gz	27.2 kB	Sunday
root.pdf	110.7 kB	Sunday
root.log	15.3 kB	Sunday
root.aux	852 bytes	Sunday
related-work.tex	0 bytes	04/10/2017
preliminaries.tex	0 bytes	04/10/2017
overview.tex	3.3 kB	04/10/2017
macros.tex	553 bytes	04/10/2017
introduction.tex	0 bytes	04/10/2017
evaluation.tex	0 bytes	04/10/2017
conclusion.tex	0 bytes	04/10/2017
approach.tex	0 bytes	04/10/2017
acmsmall-sample-bibfile.bib	0 bytes	04/10/2017
acmsmall.cls	107.5 kB	04/10/2017

Select the folder to find projects to import

Figure 6: Choose the src directory

Step 5: If you don't have the ANTLR IDE eclipse plug in already installed, go to Help → "Eclipse MarketPlace" and search for ANTLR IDE and install the ANTLR 4 IDE 0.3.6.



Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Find: ANTLR IDE

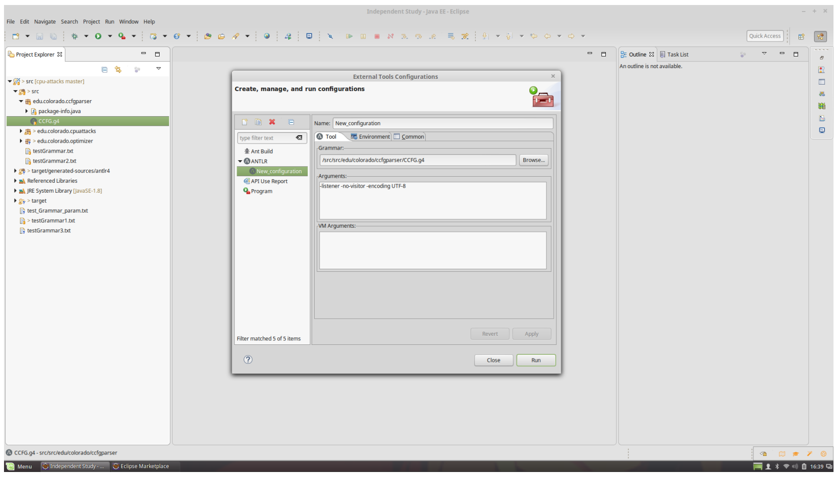
ANTLR IDE 2.1.2
ANTLR 3.0, 3.1, 3.2, 3.3 and 3.4 ANTLR Launcher and Debugger(Java only) ANTLR Built-in Interpreter Source Formatter (Ctrl+Shift+F) RailRoad diagrams Custom... more info
by Edgar Espina, EPL
IDE build editor DSL antlr
Installs: 10.5K (58 last month)

ANTLR 4 IDE 0.3.6
ANTLR 4.x Advanced Syntax Highlighting (even for target language) Automatic Code Generation (on save) Manual Code Generation (through External Tools menu) Code... more info
by Edgar Espina, EPL
antlr antlr4 antlrv4 v4
Installs: 29.0K (886 last month)
Installed

AntlrDT 4.3.1
Contributes an ANTLR V4 grammar editor and builder to the Eclipse platform. Features Antlr Editor and Outline View &ndash; full syntax-directed editor Grammar... more info
by Certiv Analytics, EPL

Figure 7: Install ANTLR IDE from Eclipse Marketplace

Step 6: Run CCFG.g4 file present under *src/edu.colorado.ccfgparser* using antlr. ***Important:*** *Make sure -visitor flag appears (and not -no-visitor).* This should automatically generate the necessary Java source files under the *target/generated-sources/antlr4/edu/colorado/ccfgparser* folder.



External Tools Configurations

Create, manage, and run configurations

Name: New_configuration

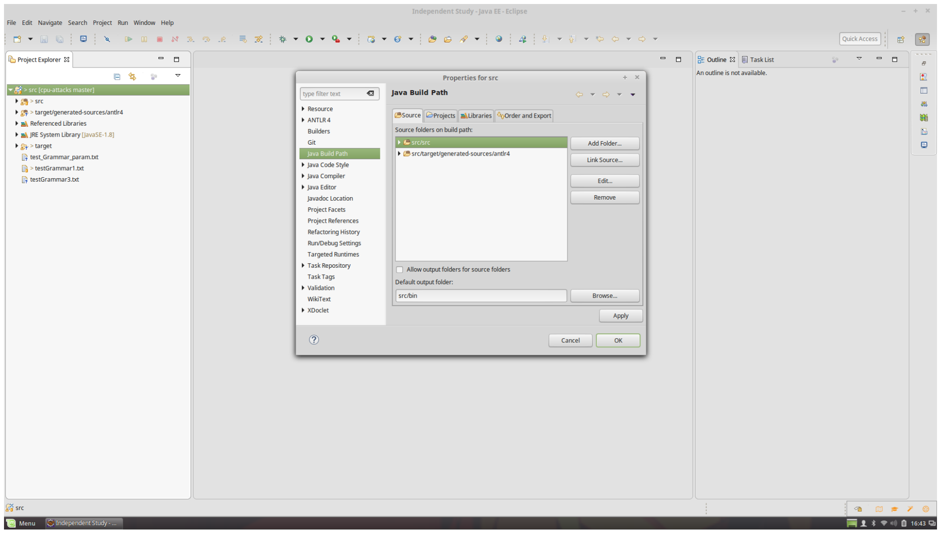
Grammar: /src/src/edu/colorado/ccfgparser/CCFG.g4

Arguments: -listener -no-visitor -encoding UTF-8

VM Arguments:

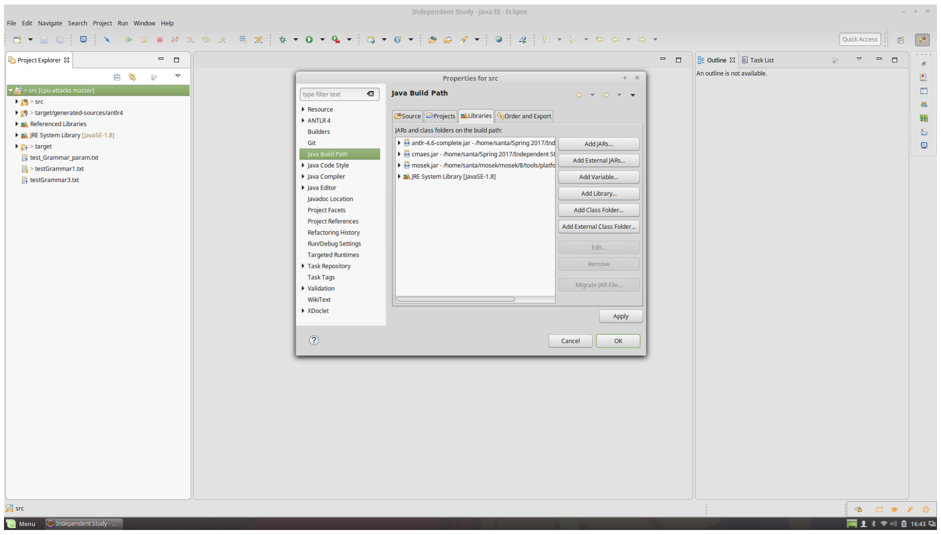
Figure 8: Set visitor flag in G4 file run configuration

Step 7: Go to Build Path→Configure Build Path by right clicking on the project.
In Sources Tab: Add the target/generated-sources/antlr4 folder
In Libraries Tab: Add the mosek.jar, antlr-4.6-complete.jar & cmaes.jar
Hit **'Apply'** and **'OK'**



Source folders on build path:
src/src
src/target/generated-sources/antlr4
Allow output folders for source folders
Default output folder:
src/bin

Figure 9: Add Antlr generate source to build path

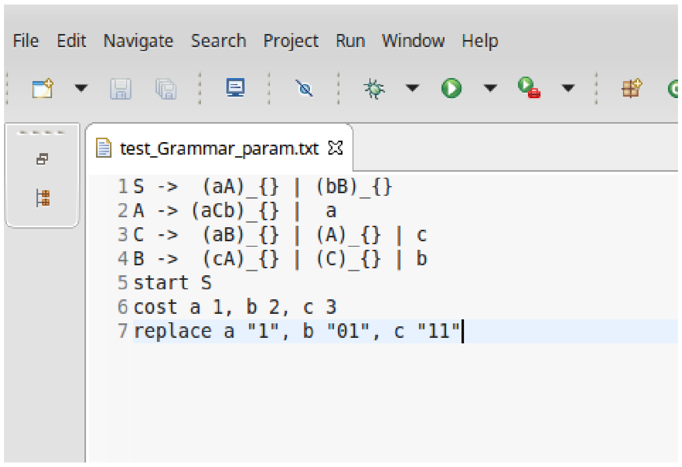


JARs and class folders on the build path:
antlr-4.6-complete.jar - /home/santa/Spring 2017/Ind
cmaes.jar - /home/santa/Spring 2017/Independent St
mosek.jar - /home/santa/mosek/mosek/8/tools/platfo
JRE System Library [JavaSE-1.8]

Figure 10: Add Jars to Build Path

Step 8: Check if there is any compile error in the code, if so fix it. Else, you can go ahead and run the code to test it as described in the next step.

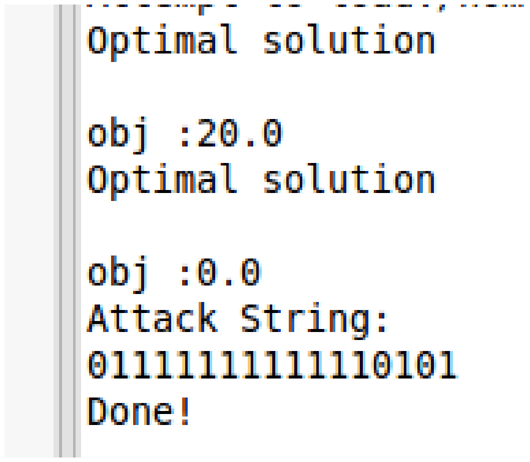
Step 9: To Run the code either run CMAES.java or SDS.java depending on what global optimization algorithm you wan to use.
Input File: The grammar that needs to be inputed to the code must be put inside a file called *"test_Grammar_param.txt"*.



```
S ->  (aA)_{} | (bB)_{}
A -> (aCb)_{} |  a
C ->  (aB)_{} | (A)_{} | c
B ->  (cA)_{} | (C)_{} | b
start S
cost a 1, b 2, c 3
replace a "1", b "01", c "11"
```

Figure 11: Sample Parameterized Input

Step 10: Check to see whether you get the desired output.



```
Optimal solution

obj :20.0
Optimal solution

obj :0.0
Attack String:
0111111111110101
Done!
```

Figure 12: Sample Output

6 Result

Tested the tool with several micro-benchmarks and then passed a context free grammar that resembles the dot parser or billion laughs attack examples. The evolutionary search over a space of n dimensions where n is the number of free parameters in the grammar.

Benchmark Grammar (Capacities = 10, Fitness Value = 20)

$$\begin{aligned} S &\rightarrow (AS)\_\{\} \mid (AB)\_\{\} \\ A &\rightarrow (aB)\_\{\} \\ B &\rightarrow b \\ \text{start } & \text{S} \\ \text{cost } & \text{a 1, b 1} \\ \text{replace } & \text{a "1", b "0"} \end{aligned}$$

The tool produced the following output 9,1,10 for this grammar suggesting us to use more of rule AS than AB to get max length string.

DOT graph Grammar (Capacities = 100 Fitness Value = 500)

$$\begin{aligned} S &\rightarrow (AA)\_\{\} \mid (SS)\_\{\} \\ A &\rightarrow (BB)\_\{\} \\ B &\rightarrow (CC)\_\{\} \\ C &\rightarrow (DD)\_\{\} \\ D &\rightarrow (EE)\_\{\} \\ E &\rightarrow e \\ \text{start } & \text{S} \\ \text{cost } & \text{e 1} \\ \text{replace } & \text{e "lol"} \end{aligned}$$

The tool produced the following output 6,5,12,24,48,96 for this grammar hinting some kind of exponential growth. Using this frequency trend we were able to produce an attack vector for the dot parser example which was one of the application that was given to us as part of DARPA Engagement challenge.

7 Conclusion

We can clearly see from the results above and from few benchmarks used earlier that this tool or technique can be of good aid to the one using it by directing him/her in a way that could help one leverage some potential bugs in the application he is looking into. Although this tool or technique works well for the given examples there are a lot of enhancements and optimizations that can be done to improve this tool like for example is there any way we could solve the problem just by using one ILP rather than two, is there a better way to tune the parameters of the evolutionary techniques, can the input be given in any different way. I hope that this report serves as a good guidance to the one continuing this work.

8 Acknowledgement

I would like to thank all people who helped me through out the semester by giving able guidance, suggestions, thoughts and being a critic when necessary. I personally would like to thank Aleksander Chakarov for his Grammatical ILP Solver tool without which any of this wouldn't been possible and assistant research professor Ashutosh Trivedi for serving as my mentor throughout this study and Assistant Professor Pavol Cerny for giving me this opportunity.

References

[1] Thummalapenta, Suresh, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Tao Xie. "Lexar: Generating String Inputs for Loop-Exploiting Attacks via Evolutionary Techniques."

[2] Bishop, J. M. "Stochastic searching networks." Proc. 1st IEE Conf. on Artifical neural networks. 1989.

[3] Moglich, M., U. Maschwitz, and B. Holldobler. "Tandem calling: a new kind of signal in ant communication." Science 186.4168 (1974): 1046-1047.

[4] Chakarov et al. Grammatical ILP Solver, 2016. Link