Grammar Learning through Symbolic Execution

By Aditya Agarwal (ANU) Supervisors: Adrian Herrera (DST), Tony Hosking (ANU)

1 Fuzzing

1.1 Fuzzing is stupid(ly inefficient)

Modern fuzzers work by passing large amounts of random input to a program to make it crash, which typically requires the availability of a large number of seeds. While there are algorithms, such AFL's T-min that achieve better outcomes that purely random seeds by mutating existing input, they still rely on random chance. This can be improved if we have access to the input grammar.

Random Fuzzing also tends to target "shallow" bugs - I.e. bugs that occur close to the start of the program. If the inputs are generated by a grammar it's parse can accept, it might be possible to find bugs deeper in the programs control flow graph.

1.2 Input Grammars are usually not available

Input specifications for many tools are simply not published. Even if a specification is published, there is no guarantee that the implementation actually follows it!

This can be alleviated by manually analysing programs, but that can be incredibly time consuming, especially when the source is unavailable.

2 Symbolic Execution

Symbolic Executors work by executing programs on abstract "symbolic" inputs. They explore the program by executing every branch in the program binary, creating a Boolean satisfiability formula for each path they go through. However, this leads to a path explosion as program complexity increases.

This project uses S2E to concolically explore simple arithmetic parsers, and build grammars from the input data.

So far, the project has succeeded in limiting path explosion, and extracting information about inputs longer than would be possible by simply brute-forcing all possible inputs.

2.1 Path Explosion

A symbolic executor generates a new state for every fork it encounters in the program. As the number of forks increases, and if a program contains loops, this leads to an exponential proliferation of states, known as path explosion.

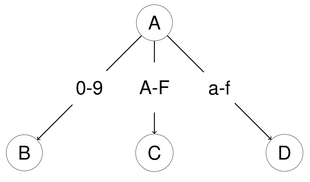


Figure 1: Consider a parser that accepts a hexadecimal digit as input. Each byte splits into three (valid) cases, corresponding to each range a hex digit can take. Because this happens for each byte considered, for a n byte string, we have 3^n states

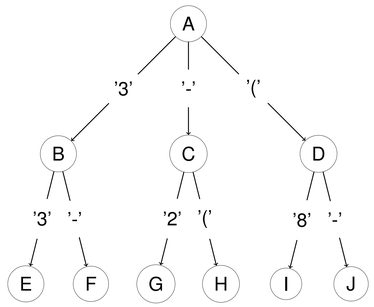


Figure 2: Consider an parser that accepts strings in the standard infix notation for arithmetic. While exploring a two byte long input, the tree might split into the cases shown above. The parser treats both "-2" and "33" as a number, so any future parsing (and subtrees generated) below nodes E and G will be identical. If we can identify such cases, we may be able to limit the path explosion.

3 Limiting Path Explosion

3.1 Identifying duplicate states

We extract data by brute-forcing small inputs, identifying "equivalent" inputs, and only executing one path from.

Inputs i, j (and their corresponding states) are equivalent if for all strings s, is is accepted by the parser iff js is.

To achieve this, we first construct a prefix tree acceptor, that accepts exactly the inputs we have seen.

Clearly, If the subtree rooted at any two nodes in the (potentially infinite) acceptor tree for the language are isomorphic, then they are equivalent.

However, identifying this is not possible with finite data so we approximate this by guessing that nodes are equivalent if they have the same inputs, and they have identical transitions. E.g. the nodes E and G in Figure 2.

We do not consider isomorphic finite subtrees to be a sufficient critera because otherwise we would merge all childless accepting states. E.g. "3+3" and "(3)" for a 3 byte tree!

4 Exploring Longer Inputs

By exploring only one path from each equivalence class of valid inputs, we can greatly limit the number of paths we explore. However, this proves overly restrictive – there may be inputs of length n that are not accepted but are parts of longer inputs that are accepted. E.g. "(2+3" is not accepted, but is part of the longer (valid) input "(2+3)".

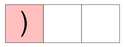
To account for this we consider "experimental" sentences.

4.1 Experimental States

Experimental sentences are the sentences s such that s itself is not a valid sentence in the language, but might be a prefix of some valid sentence s'.

We identify them by checking whether the parser inspected every byte of s before rejecting the corresponding state, as in 3a.

(a) The string "(13" is not a valid sentence in our grammar, but it is the prefix to the valid sentence "(13)", so the parser inspects all bytes.



(b) However ")" is not a valid sentence in our grammar, nor is it the prefix of any valid sentence, so the parser rejects it before inspecting every byte.

(c) We might also have a case where the sentence is invalid, and is not the prefix of any valid sentence, so it will not spawn any child nodes.

Figure 3: Consider a LL(1) parser for arithmetic. It will inspect a byte m if and only if the tokens it has already inspected are valid prefixes. In this figure, red represents bytes the parser has inspected, and white the bytes the parser hasn't.

We can reject cases where the parser did not inspect every single byte of s, because most parsers will parse the bytes of a file in the same order, thus rejecting all inputs containing s as a prefix, as in 3b.

We perform a similar analysis on the experimental set as the acceptor tree to identify equivalent states and reduce the number of duplicate paths we traverse.

Then we expand each of experimental nodes and accepting nodes.

This is done by constructing a concrete representative input for some input that would lead to that state.

If an experimental node has no valid or experimental children (I.e. no child where every byte was inspected) then we remove that node. This accounts for cases where the last byte made the sentence invalid, as in 3c.

5 Results

This refinement has allowed us to explore significantly larger state spaces, e.g. allowing us to explore all distinct input classes upto 6 bytes in length in 10 minutes, for an hexadecimal arithmetic parser, while the same machine would be unable to finish 4 bytes in >12 hours.

6 Future Work

6.1 Better heuristics

- The deduplication heuristic has a large false negative rate, missing expressions that are equivalent, like "33 " and "333" for an arithmetic parser. (Both get parsed to INT).
- It also assumes that the parsers are "well behaved". We can always define a language where the subtree appears isomorphic upto a finite number of bytes - so our heuristic would create false positives, merging non-equivalent states.

6.2 Control-flow Analysis

- S2E can log the path of the program being executed on certain inputs.
- This gives us some information on the internal structure of the parser, which can be used to better guess an automaton that corresponds to the program.
- We can then use directed symbolic execution [5] to generate test cases (and their corresponding constraints).
- We may also be able to augment the deduplication heuristic. Preliminary work shows that it is possible to detect duplicates the state-merging heuristic misses, however this method loses semantic information and has other false negatives. It should be possible to combine the two heuristics, to improve deduplication.

6.3 Building Regular and Context Free Grammars

- We can generate positive and negative examples of strings our language accepts.
- It should be possible to use this to inductively define finite automata that accept strings in the language, including those not already seen.[7]
- This can then be used to generate test cases that achieve wider or more targeted coverage of the input parser. [6]
- If we are able to extract structural data using control flow analysis, then we may be able to use it to efficiently infer context free grammars. [8]

References

[1] Bastani, O., Sharma, R., Aiken, A. and Liang, P. (2017). *Synthesizing program input grammars.* ACM SIGPLAN Notices, 52(6), pp.95-110.

[2] Botinčan, M. and Babić, D. (2013). *Sigma**. ACM SIGPLAN Notices, 48(1), pp.443-456.

[3] Chipounov, V., Kuznetsov, V. and Candea, G. (2012). *The S2E Platform*. ACM Transactions on Computer Systems, 30(1), pp.1-49.

[4] Cui, W., Peinado, M., Chen, K., Wang, H. and Irun-Briz, L. (2008). *Tupni*. [online] Available at: https://dl.acm.org/citation.cfm?doid=1455770.1455820 [Accessed 18 Jan. 2018].

[5] Ma, K. (2011). *Directed symbolic execution*. [ebook] Available at: https://drum.lib.umd.edu/bitstream/handle/1903/11374/CS-TR-4979-r1.pdf?sequence=3&isAllowed=y [Accessed 18 Jan. 2018].

[6] Majumdar, R. and Xu, R. (2007). *Directed test generation using symbolic grammars*. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07.

[7] Angluin, D. (1987). *Learning regular sets from queries and counterexamples*. Information and Computation, 75(2), pp.87-106.

[8] Sakakibara, Y. (1990). *Learning context-free grammars from structural data in polynomial time.* Theoretical Computer Science, 76(2-3), pp.223-242.

Australian National University

ANU College of Engineering & Computer Science