CISPA Helmholtz-Zentrum i.G.

Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing

Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller

{rahul.gopinath, bjoern.mathis, hoeschele, kampmann, zeller}@cispa.saarland
CISPA / Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

CISPA
HELMHOLTZ-ZENTRUM i.G.

Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing

Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller
{rahul.gopinath, bjoern.mathis, hoeschele, kampmann, zeller}@cispa.saarland
CISPA / Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

(Dated October 18, 2018)

Abstract Generating valid test inputs for a program is much easier if one knows the input language. We present first successes for a technique that, given a program P without any input samples or models, learns an *input grammar* that represents the syntactically valid inputs for P—a grammar which can then be used for highly effective test generation for P. To this end, we introduce a *test generator targeted at input parsers* that systematically explores parsing alternatives based on dynamic tracking of constraints; the resulting inputs go into a *grammar learner* producing a grammar that can then be used for fuzzing. In our evaluation on subjects such as JSON, URL, or *Mathexpr*, our PYGMALION prototype took only a few minutes to infer grammars and generate thousands of valid high-quality inputs.

1 Introduction

Testing programs with generated inputs is a common way to test programs for robustness. Such generated inputs must be *valid,* because otherwise, they would be rejected by the program under test before reaching the functionality to be tested; and they must well sample *the full range of possible inputs,* because otherwise, important program features may not be covered. In the absence of a formal input specification such as a grammar, common test generators have to rely on *samples of valid inputs.* These would then 1. be systematically mutated [15] using generic operations such as bit flips or character exchanges; or 2. be used to *infer grammars and syntactical rules* that can then be used to generate more similar inputs [10, 1, 7]. Both approaches, however, would have great difficulty synthesizing features that are not present in the original samples already. In principle, test generators could use *symbolic analysis* on the program under test to determine and solve the exact conditions under which an input is accepted [14, 12, 3, 4]; but nontrivial input formats induce a large number of constraints that can easily overwhelm symbolic constraint solvers.

In this paper, we follow recent advances in grammar inference [10, 1, 7] by first learning an input grammar, and then using this grammar for test generation. In contrast to this state of the art, however, our approach automatically infers an accurate description of the input language *without requiring any input samples at all*—actually, all that is needed for comprehensive testing is the program itself. Figure 1 summarizes our approach:

1. To address the problem of learning without samples, we introduce a *test generator specifically targeting input parsers.* Our approach starts with a fixed input (typically an empty string), which would be rejected. During parsing, we use dynamic tainting[1] to dynamically track all comparisons of input characters against expected values, and then provide an input that satisfies these expectations. By repeating this process from rejection to rejection, we eventually obtain a set of inputs that covers all comparison alternatives made by the parser—and consequently, all structural (syntactic) alternatives as well.

2. To *learn a grammar* from the parser-covering inputs, we dynamically track the *data flow of input characters* throughout program execution to induce a grammar. Our main algorithm is inspired by Hoschele et al. [10]: Character sequences that share the same data flow then form syntactic entities; subsequences with different data flow induce composition rules. On top, our grammar learner makes use of *equivalence classes* found during Step 1: If the test generator finds that, say, some input fragment can be any digit, this generalization is also reflected in the grammar.

3. To *produce inputs,* we use the grammar from Step 2 as a *producer,* now very rapidly producing inputs for the program under test. At this point, no instrumentation of the program is required anymore, and the inputs produced could also be given to another program with the same input language.

As a result, we obtain a tool chain that *requires nothing but an executable code, and produces high-quality inputs that cover and combine all syntactic features.* Our PYGMALION prototype[2] for Python programs requires only a few minutes to infer accurate grammars and produce thousands of valid inputs for formats such as JSON. Our approach is generic in its use of tools, as we could easily integrate different grammar learners or producers. It is also versatile in its purposes, as the resulting grammars could also

[1] We use a *pure python* library similar to the algorithm by Conti et al. [5] for tracing taints and comparisons. Hence, our algorithm *does not* require specially crafted interpreter to track taints.

[2] PYGMALION = PYthon Grammar Miner Actively Learning Inputs Of Note. See also Pygmalion [13] on language learning..

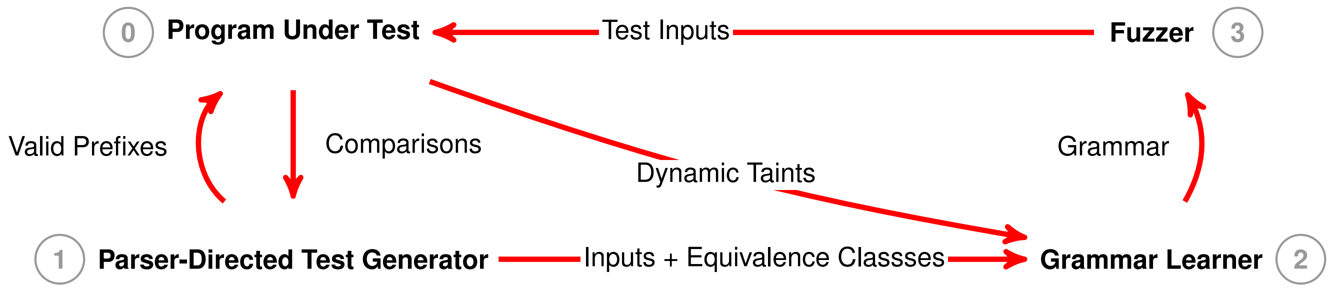

Figure 1: The PYGMALION prototype starts with a program under test (0) into which we feed a fixed, valid prefix (say, an empty string). By dynamically tracking the comparisons of input characters against expectations, a *parser-directed test generator* (1) systematically satisfies these expectations, eventually producing a set of inputs that cover all parser alternatives. These go into a *grammar learner* (2), which by tracking the data flow of these characters through the program produces an input grammar. Using this grammar, a *fuzzer* (3) can now produce syntactically valid program inputs at high speed, systematically covering input features.

be used for activities such as input understanding, program understanding, parsing and translating inputs, or debugging.

The remainder of this paper is organized as follows. Section 2 illustrates our approach using arithmetic expressions as an example, devoting a section each to the individual steps from Figure 1:

1. Section 2.1 details how we generate inputs to systematically cover parsing alternatives.
2. Section 2.2 shows how we use the resulting inputs and equivalence classes to induce high-quality grammars.
3. Section 2.3 discusses how we use these grammars as producers, reentering grammar induction should generated inputs be rejected.

Section 3 evaluates our publicly available PYGMALION prototype testing formats such as URL and JSON. We find that PYGMALION achieves the same coverage as constraint-based alternatives; its inputs, however, are not only much more likely to be valid, they also cover and combine more features of the input language. Section 4 closes with conclusion and future work.

2 Our Approach in a Nutshell

To illustrate our approach, let us assume we want to exhaustively test some mystery program P. We know nothing about P; in particular, we have no documentation or example inputs. What we know, though, is that 1. P accepts some input I sequentially as a string of characters; and that 2. P can tell us whether I is a valid or an invalid input. We further assume that we can *observe* P processing I: Specifically, we need to be able to observe the dynamic *data flow of input characters* from I as P processes them.

2.1 Step 1: Testing a Parser

In Step 1 (Figure 1), we explore the capabilities of P's input parser by means of directed test generation. The key idea is to observe all *comparisons an input character goes through*, and systematically satisfy and cover alternatives, notably on rejection.

We start with an empty string as input, which is rejected as invalid immediately as *EOF* is encountered. The *EOF* is detected as any operation that tries to access past the end of given argument. This error is fixed in the next round by testing P with a random string, say `"A"` (I = `"A"`). Indeed, this input is also rejected by P as invalid. Before rejecting the input, though, P checks I for a number of properties:

1. Does I start with a digit?
2. Does I start with a `'('` character?
3. Does I start with `'+'` or `'-'`?

Only after these checks fail does P reject the input.

All these conditions are easy to satisfy, though—and this is a *general* property of parsers, which typically only consider the single next character. Using a combination of depth-first and breadth-first search, our test generator picks one condition randomly. Satisfying Item 1, it would produce a digit as input (say, `"1"`). This would now be accepted by P as valid, and we have generated our first input.

After the acceptance of `"1"` as a partial input, P conducts a check to see if another character follows `"1"` by accessing the next character in the input. Since P reached the end of the string we consider the prefix as valid and add another random character. This results in the new prefix `"1B"` which results in new conditions: Is the `"B"` a digit? Or any of the characters `'+'`, `'-'`, `'*'`, or `'/'`? Again, one of these conditions is chosen randomly, together with the prefix `"1B"` seen so far.

In a consecutive execution with another random seed, the first condition to be addressed might be Item 2. Satisfying this condition yields `"("` which will again cause the parser reaching the end of the input, so we append a random character and get `"(C"` as input. This is rejected, but only after again checking for a number of expected characters that could follow. These would be the same checks already performed on the input `"A"`: digits, parentheses, `'+'`, and `'-'`. We randomly choose the condition Item 3, where again the prefixes `"(+"` and `"(-"` would be invalid on their own, so we again choose one prefix for further computations.

By continuing this process, we thus obtain more and more inputs that systematically cover the capabilities of the parser. In the end, we obtain a *set of legal inputs that covers all the conditions encountered during parsing*:

```
1   11   +1   -1   1+1   1-1   1*1   1/1   (1)
```

We see that our mystery program P in fact takes arithmetic expressions as inputs.

2.2 Step 2: Inducing a Grammar

In Step 2 (Figure 1), we take the generated inputs together with P to induce an *input grammar*—that is, a context-free grammar which describes the input language of P. To this end, we feed the generated inputs into P while *tracking their data flow,* notably into variables and function arguments.

We find that an input such as `"1+1"` flows into a function `parse_expr()`, which 1. first recursively invokes `parse_expr()` on the left `"1"`, 2. then invokes `parse_binop()` on the `"+"`, and 3. finally recursively invokes `parse_expr()` on the right `"1"`. Tracking the recursive calls of `parse_expr()` on `"1"`, we find that these invoke `parse_int()`, which in turn invokes `parse_digit()`, always passing the `"1"` as argument. From this sequence of calls, we can now induce a grammar rule, using these key ideas:

1. First, we can associate input fragments with the functions that successfully process them and assume that each input argument to a function represents a syntactic entity. Hence, `"1"` is a *digit,* an *integer,* and an *expression;* `"+"` is a *binary operator;* and `"1+1"` is an *expression.*

2. Second, if some entity E is a substring of a larger entity E', we can derive a grammar rule decomposing E into E'. In the above case, we obtain rules such as

   Expr $\rightarrow$ *Int* | *Expr BinOp Expr*;
   BinOp $\rightarrow$ **"+"**;
   Int $\rightarrow$ *Digit*
   Digit $\rightarrow$ **"1"**;

3. Third, during parser-directed test generation, we track *equivalence classes* as induced by successful conditions. We thus know that besides `"1"`, any digit would have satisfied the conditions seen. We can thus replace `"1"` with the equivalence class of all digits:

   Digit $\rightarrow$ `/[0-9]/`;

4. Finally, we can repeat the process for all inputs seen during the parser-directed test generation in Step 1. This introduces *alternatives* for all elements processed in the grammar, covering all operators and other syntactic features. The resulting grammar (Figure 2) represents all alternatives seen.

With this, we now have obtained a full description of P's input language—without any sample inputs, specification, or model.

2.3 Step 3: Grammar-Based Fuzzing

Grammars as obtained in Step 2 can serve many purposes. We can use them to understand the structure of inputs, as well as the programs that process them. We can use them to parse and process existing inputs, for instance to create detailed statistics on the occurrences of specific elements, or to protect programs against invalid inputs. Our main application in this paper, though, is their use for *test generation.*

Turning a grammar into a producer is a simple exercise. Starting with the start symbol (*Expr* in our case), we keep on replacing nonterminal symbols by one of the alternatives until only terminal symbols are left. To avoid boundless expansion, we can set a limit on the maximum length of the string; once this is reached, we always prefer expansion paths that lead to terminal symbols.

This generation process now no longer requires any execution, instrumentation, or analysis of the program under test. Hence, it is fast; and the strings generated can even be applied to some other program P' that shares the input language with P. A simple grammar producer can thus easily generate thousands to millions of inputs per minute, covering all kinds of symbols and their combinations. This is what our technique produces: *Given only a program P, without any input samples, we obtain an input grammar that accurately describes the input language of P, and consequently, can generate as many syntactically valid test inputs as desired.*

3 Initial Results

We have implemented the above approach as a proof-of-concept prototype in Python, named PYGMALION. We evaluate PYGMALION and all its parts on three different formats: JSON [9], URL[3], and mathematical expressions [11]. We used a coverage tool for Python [2] to compute the coverage the inputs achieve on the different subjects. For comparison, we used the AFL [15] random fuzzer and KLEE [3], a symbolic execution engine, both state of the art input generators. Since KLEE is not available for Python, we generated inputs with KLEE on a C parser of the respective input language and then executed the Python parser for this language with the generated inputs. AFL and KLEE were both run with default settings. Table 1 summarizes our results, detailed in the remainder of this section.

[3] We manually converted the Java URL parser [6] to Python.

```
Expr → Int | UnOp Expr | Expr BinOp Expr | "(" Expr ")";
UnOp → "+" | "-";
BinOp → "+" | "-" | "*" | "/";
Int → Digit+
Digit → /[0-9]/;
```

Figure 2: Grammar induced from the inputs in Step 1

```
-++7 / +(9 - 6 / 7 + 5) - 1 + (0) / -75 * +(3 - 6 - 0 - 7)
9 + 4 - 3 + 7 / 7 + 3 / +3 * (9 - 2 * 9) - 8
++--+7 - (6 * 6 * 3) / (0 + 2) / +(5 / 6 / 5 + 3 * 1)
3 * 2764 + 1 / 0 * 4 / -5 / 6 * (1 * (8) + 9 / 4 * 0 * +4)
3 * 5 + 0 * 0 / 8 - 7 * 7 * ++(2 + 5 - 9 * 9)
+05834 * --(46 + +1 / +-+(-46 / 4) - --(63 - -(5 + 1 + +2 * 0 /
    ++82 + (9 + 6)))) / -404471632
3 - 4 / 5 - 0 / 6 + 1 * 9 * 4 - +334
8 + 7 / 4 * 9 - (3 + 6 - (7)) + -0 - -+(5 + 8) * -++++5 - -2973
```

Figure 3: Fuzzing output from the grammar in Figure 2

3.1 Execution Time

We let parser-directed test generation run until it produced 100 inputs. The length of inputs produced by parser-directed test generation is affected by the complexity of input grammar. In particular, when considering nested grammars, each successive character might increase the amount of nesting in the string produced, by adding a character—e.g. '('—or close existing nested structures—e.g. ')'. Since we are interested in valid strings, after a fixed number of characters is produced, we switch to a strategy designed to identify short suffixes that can complete the current string prefix. The inputs from parser-directed test generation was used to infer the grammar (Steps 1 and 2); we then used this grammar to produce 1,000 inputs (Step 3). For producing samples from the grammar, we chose to limit the number of symbols expanded to 100 before applying heuristics to complete the string generation.

Table 1 reports the PYGMALION execution times broken down per step; Steps 1 and 2 need to be run once per program, Step 3 for every 1,000 inputs generated. Note that switching from Python to C would speed up all three steps further, especially Step 3.

For comparison, we let AFL and KLEE run as long as all three phases of PYGMALION and assessed the resulting test cases. AFL has no built-in limit to how long it will run and produce inputs; KLEE stops as it has explored all paths, but would not reach this limit within the execution time of PYGMALION.

3.2 Input Validity

For all three subjects, between 73% and 78% of all inputs generated by PYGMALION would be valid; the remainder is invalid due to overgeneralization in Step 2. For AFL, we only report those inputs where it found a new path (which is the default setting); only between 0% and 50% of these inputs, though, are valid. KLEE produced thousands to millions of inputs, with 25% to 46% being valid. Most of the inputs of AFL and KLEE exercise handling of syntax errors.[4]

3.3 Coverage

Let us now come to the one metric typically used to compare the performance of test generators—coverage. We only report coverage of code handling valid inputs, as this would be the code that actually holds program functionality. (As discussed before, if one wanted to deliberately produce invalid inputs, AFL would probably be the best choice.) PYGMALION and KLEE achieve a very similar coverage. The only 1-point difference is in URL, where KLEE explores URL queries (prefixed by '?') and PYGMALION doesn't; the reason is that (a) the URL parser accepts any

[4]For URL, actually *none* of the inputs generated by KLEE would be valid in the original Python subject because the C subject we applied KLEE on would erroneously accept URLs without a protocol prefix. For fairness, we therefore changed the Python parser to also accept URLs without prefix.

Table 1: Results of fuzzing with valid inputs of PYGMALION, AFL, and KLEE.

Language	Time	# Valid Inputs/# Inputs			Statement Coverage			Maximal Input Length		
	PYG (Step 1+2+3)	PYG	AFL	KLEE	PYG	AFL	KLEE	PYG	AFL	KLEE
URL	66s + 53s + 28s	782/1,000	0/36	463,165/1,028,735	55%	0%	56%	275	0	31
Mathexpr	96s + 1,7s + 18s	736/1,000	14/80	54,867/498,801	63%	50%	63%	200	23	16
JSON	98s + 19s + 9s	778/1,000	19/39	12,575/41,625	43%	23%	43%	81	29	31

AFL and KLEE were given the same time as PYG for all subjects string after the hostname, with no special provisions for `'?'` (queries) or `'#'` (anchors); and (b) PYGMALION's grammar inference does not generalize the characters to include `'?'` characters. Apart from `'?'`, in all three cases, the coverage achieved by PYGMALION is the maximum one can achieve on these subjects using valid inputs.

3.4 Input Quality

A good test case will not only cover code, but also explore *combinations* of features to thoroughly test their possible interactions and interference. As a very simple assessment of how our inputs fare in this regard, we take a look at the generated valid inputs with *maximum length*, for example for JSON:

- The longest PYGMALION input covers and combines JSON elements such as arrays, objects, strings, and numbers[5]:

```
[false ,[{  "o":{    ,   "$dYPrlj@?BR":
 [+ ]"S|+|4GzCW(C":-94}}  ],
 [false,null]]
```

- The longest AFL input consists of 29 periods (`'.'`)
- The longest KLEE input consists of the keyword `null`, followed by 27 8-bit ASCII 255 (`'ÿ'`) characters.

We see that the PYGMALION input is *considerably richer in syntax and semantics.* For *Mathexpr*, the situation is the same: AFL and KLEE produce a long single number, whereas PYGMALION combines elements as in Figure 3; only for URL does the longest KLEE input actually cover elements of the URL structure.

One may argue that PYGMALION is set to produce 100 symbols and thus longer inputs than KLEE with 30 characters per input[6]. But then, the search effort reduces for KLEE if the input size is small while still getting the chance to produce complex inputs. But even with 30 characters KLEE is not able to produce any complex inputs that make use of the size. Furthermore, and this is precisely the point: When producing from a grammar, not only do we get well-structured complex inputs, as with PYGMALION. *For a tester, it also is very easy to control input length or depth, to emphasize or de-emphasize symbols, or to favor or cover specific combinations of symbols.* This is only possible with well-structured and well-readable grammars, whose inference thus contributes to the quality of test cases and the potential of our approach.

4 Conclusion and Consequences

We have shown that it is possible to *determine the input language from a given program alone, without requiring input samples.* This finding has many applications throughout programming languages and software engineering, for instance in understanding both input and program structure. First and foremost, though, this is an important step forward for test generation at the system level, which so far required either 1. a model for the input (say, an input grammar or a state model), or 2. a set of sample inputs (which would be mutated, evolved, or abstracted into a model). In contrast, our approach makes it possible to take a given program only, infer its input language automatically, and immediately use this for producing syntactically valid inputs with high coverage—all without any human effort, as we demonstrate in this work. At the same time, the inferred grammars give testers (and tools) control over which and how many elements should be covered and generated, targeting features and feature combinations much better than mutation-based or constraint-based approaches.

While Python is a great language for prototyping, most common input formats are parsed in C programs and libraries; therefore, we are currently implementing our approach for C programs. In the meantime, a replication package with PYGMALION source code as well as all experimental settings and data is available [8].

[5] It also exercises a bug in the *microjson.py* parser.

[6] AFL can generate inputs of arbitrary size.

References

[1] Osbert Bastani et al. "Synthesizing Program Input Grammars". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 95–110. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062349. URL: http://doi.acm.org/10.1145/3062341.3062349.

[2] Ned Batchelder. *Coverage Python*. https://coverage.readthedocs.io/en/coverage-4.5.1/. 2018.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[4] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *Proc. 39th IEEE Symposium on Security and Privacy*. San Francisco, California, USA, 2018. URL: http://arxiv.org/abs/1803.01307.

[5] Juan JosÃl' Conti and Alejandro Russo. "A Taint Mode for Python via a Library". In: *Proceedings of the 15th Nordic Conference on Information Security Technology for Applications*. NordSec'10. Espoo, Finland: Springer-Verlag, 2012, pp. 210–222. ISBN: 978-3-642-27936-2. DOI: 10.1007/978-3-642-27937-9_15. URL: http://dx.doi.org/10.1007/978-3-642-27937-9_15.

[6] Oracle Corp. *URLStreamHandler.java*. http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/net/URLStreamHandler.java. Accessed: 2018-04-13. 2018.

[7] Patrice Godefroid, Hila Peleg, and Rishabh Singh. "Learn&Fuzz: Machine Learning for Input Fuzzing". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 50–59. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155573.

[8] Rahul Gopinath. *Pygmalion repository*. https://github.com/vrthra/pygmalion. Accessed: 2018-04-13. 2018.

[9] Patrick Hensley. *Minimalistic Python JSON parser, small enough to be embedded into standalone scripts*. https://github.com/phensley/microjson. Accessed: 2018-04-13. 2018.

[10] Matthias HÃűschele and Andreas Zeller. "Mining Input Grammars from Dynamic Taints". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 720–725. ISBN: 978-1-4503-3845-5. DOI: 10.1145/2970276.2970321. URL: http://doi.acm.org/10.1145/2970276.2970321.

[11] Nerd Paradise. *How to Parse Mathematical Expressions*. https://nerdparadise.com/programming/parsemathexpr. Accessed: 2018-04-13. 2018.

[12] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: ACM, 2005, pp. 263–272. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081750. URL: http://doi.acm.org/10.1145/1081706.1081750.

[13] Bernard M. Shaw. *Pygmalion*. New York, NY, USA: Brentano's Press, 1916.

[14] Willem Visser, Corina S. PÇŐsÇŐreanu, and Sarfraz Khurshid. "Test Input Generation with Java PathFinder". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 97–107. ISBN: 1-58113-820-2. DOI: 10.1145/1007512.1007526. URL: http://doi.acm.org/10.1145/1007512.1007526.

[15] Michal Zalewski. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/. Accessed: 2018-01-28. 2018.