Source Code Implied Language Structure Abstraction through Backward Taint Analysis

Zihao Wang[1], Pei Wang[2], Qinkun Bao[2], and Dinghao Wu[1]
[1]*Pennsylvania State University, University Park, USA*
[2]*Individual Researcher, USA*
zihao@psu.edu,{uraj, qinkun}@apache.org, dinghao@psu.edu

Keywords: Program Analysis, Context-free Grammar, Static Analysis, Fuzzing, Data-flow Analysis, Taint Analysis

Abstract: This paper presents a novel approach for inferring the language implied by a program's source code, without requiring the use of explicit grammars or input/output corpora. Our technique is based on backward taint analysis, which tracks the flow of data in a program from certain sink functions back to the source functions. By analyzing the data flow of programs that generate structured output, such as compilers and formatters, we can infer the syntax and structure of the language being expressed in the code. Our approach is particularly effective for domain-specific languages, where the language implied by the code is often unique to a particular problem domain and may not be expressible by a standard context-free grammar. To test the effectiveness of our technique, we applied it to libxml2. Our experiments show that our approach can accurately infer the implied language of some complex programs. Using our inferred language models, we can generate high-quality corpora for testing and validation. Our approach offers a new way to understand and reason about the language implied by source code, and has potential applications in software testing, reverse engineering, and program comprehension.

1 Introduction

Modern software can be thought of as abstract machines that operate on a set of symbols, similar to how the Turing machine operates on its tape. In computer science, the set of symbols used by a program is known as a language, which can be represented by a formal grammar. A grammar is a set of rules that define the structure of a language and how symbols can be combined to form valid sentences.

Source code implied language refers to the language that is implicitly defined by the source code of a program or system. This language includes, but is not limited to, program input and output formats, domain-specific languages (DSLs), and communication protocols between different components or systems. In contrast to a general-purpose language, source code implied language is often tailored to a specific domain or problem space and is often characterized by a specific syntax, grammar, and terminals.

Despite that Source code implied language are critical in many ways, they are not always as available as the programs themselves. In some cases, their grammars are available, but in a form that is not friendly to computers, such as the Adobe PDF specification which is defined in a 700-page document (for Standardization (ISO) (2008); Adobe Systems Incorporated (2000)) described in human language rather than a well-defined grammar. In some other cases, there are no formal specifications for the input language at all, like the LLVM Intermediate Representation.

In this paper, we propose a static analysis algorithm for extracting implicit grammars from program source code. Our key insight is that object-oriented programming languages typically define an implicit language with classes. These classes may contain encoders, such as a parser, or decoders, such as a printer, which are often written in easily comprehensible patterns. By analyzing their implementation, it is possible to extract a grammar that represents a subset of the language that the program operates on.

Our static analysis algorithm is based on static data-flow analysis. To extract accurate and high-quality grammars, we perform context- and path-sensitive taint analysis. We made several carefully calibrated trade-offs in the analysis to improve precision while maintaining scalability when analyzing common code patterns found in target class.

We implement the above grammar extraction algorithm in a prototype and apply it on programs that can be compiled into LLVM IR to infer the target grammars automatically. To evaluate the precision of the abstracted grammar of pretty printers, we collect several small printer programs and libxml2' as test cases, and get expected results.

Our research makes the following contributions:

- We propose a novel algorithm for implicit grammar abstraction that only requires the source code of the program generating the language. To the best of our knowledge, our work is the first to achieve implicit grammar extraction without requiring access to program corpus or program input as grammar. Our approach is capable of abstracting both lexical and syntactical structures, as well as identifiers.
- We introduce a static analysis method that infers the possible values of string variables at each program point, which is crucial for generating valid inputs to a program.
- We present a prototype, which implements our algorithm and can produce readable grammars and valid corpus for languages such as XML.

Overall, our contributions provide an effective and efficient way to extract program grammars and generate valid inputs, which can significantly enhance the testing and security analysis of programs.

2 Background

In recent years, there has been a growing interest in developing machine-understandable grammars that can be processed and analyzed by computer programs (Harkous et al. (2020); Ammons et al. (2002); Gopinath et al. (2020); Lin and Zhang (2008)). These grammars are designed to be easily interpreted by software and can be used to automate a wide range of tasks, such as natural language processing, code generation, and data validation. By using machine-understandable grammars, developers can increase the efficiency and accuracy of their software and improve its ability to interact with other systems. In this paper, we explore the use of machine-understandable grammars for program output abstracting through backwards taint analysis, a novel technique for program output analysis that combines symbolic execution and dynamic taint analysis.

Machine-understandable grammars have numerous applications in software engineering, including efficient test generation. Over the decades, this technology has been extensively researched (Maurer (1990); Sirer and Bershad (1999); Coppit and Lian (2005)). Take input grammar as an example. Fuzz testing and random testing are some automatic software testing techniques that generate random, invalid, or unbiased inputs to a program to reach abnormal target states. Utilizing the structure of inputs is crucial to improving the success rate and efficiency of these technologies (Chen et al. (2021); Zhong et al. (2020); Gopinath et al. (2020); Wu et al. (2019); Toffola et al. (2017); Wang et al. (2017); Yang et al. (2011)). Without knowledge of the input structure, testing methods often remain limited to the input-checking or parsing stage, failing to achieve improved test coverage. Recent advancements in generating input corpus for fuzzing have leveraged input grammars and grammar-aware mutation algorithms, resulting in significant improvements in fuzzing efficiency and coverage for specific targets.

2.1 Mining Input Specifications

Input grammar, as a form of code implied language, provides a description of the syntax and structure of the inputs expected by a program. It serves as a fundamental type of source code implied language, making the understanding and learning of a program's input language an active research area. The related work mentioned in this section addresses related challenges and offers valuable insights that contribute to our approach.

Ammons et al. presents a machine learning approach called specification mining for automating the process of discovering formal specifications of protocols that code must follow when interacting with an application program interface or abstract data type (Ammons et al. (2002)). The approach infers a specification by observing program execution and summarizing frequent interaction patterns as state machines that capture both temporal and data dependences.

Lin et al. present the first work in extracting input grammar from programs with dynamic analysis approaches (Lin and Zhang (2008); Lin et al. (2010)). They identify most programs' input grammar into two categories: top-down and bottom-up grammars, and perform runtime analyses for each type. They perform the top-down grammar analysis based on dynamic program control dependence and the bottom-up grammar analysis based on parsing stack track. By doing this, Lin et al.'s work can handle some large-scaled programs with white-box access. However, their work requires massive manual analyses and modifications to the targets.

Höschele and Zeller use dynamic tainting to trace the data flow of sample inputs and present their prototype AUTOGRAM (Höschele and Zeller (2016)).

AUTOGRAM defines input elements which follow the same data flow as one syntactic entity. By doing this, Höschele and Zeller's method can identify functions related to input processing and further infer all the possible syntactical entities handled by the related functions. AUTOGRAM can address each grammar component to its corresponding variables in target programs, providing intuitional insights to following reverse engineering. Nevertheless, as the authors mentioned, the grammar AUTOGRAM learned is highly dependent on the given sample space. When the grammar grows, it very possible to misses some corner cases.

Furthermore, Wu et al. (2019) present REINAM, a reinforcement-learning approach to synthesize input grammar. Their two-phase approach includes: first using dynamic symbolic execution and satisfiability modulo theory (SMT) solver to obtain the program input grammar (Tillmann and de Halleux (2008); Xie et al. (2009)), and second generating a probabilistic context-free grammar (PCFG) with the help of GLADE.

2.2 Grammar-Assisted Fuzzing

When fuzzing programs that take structured inputs, coverage-based fuzzers often use grammar-sensitive approaches to increase the coverage of the inputs they test. These approaches can be classified into three categories: grammar-based mutation (Holler et al. (2012); Veggalam et al. (2016); Guo (2017); Groß (2018); Zhong et al. (2020); Chen et al. (2021); Wang et al. (2019)) grammar-based generation (Ruderman (2007); Valotta (2012); Aschermann et al. (2019); Yang et al. (2011); Godefroid et al. (2017)), and program input synthesis without knowledge of the input structure (Godefroid et al. (2008); Wang et al. (2017)).

3 Method

This section introduces our grammar-extracting algorithm. We will first present an overview of our approach and then describe some important details of the algorithm.

3.1 Overview

We first identify the source and sink functions of our analysis target using human knowledge. Then, we perform forward taint analysis based on the identified functions to extract all the functions that are transitively called by the source functions and call the sink functions. This provides us with the necessary data flow for abstracting the output grammar. We represent the source functions and tainted functions as nonterminal tokens, and primitive variables as terminals, in order to present the EBNF grammar. To improve the presentation of the grammar, we use a strongly regular syntax to approximate the abstracted grammar. Finally, we obtain a EBNF grammar of the output context represented by a series of production rules, which use regular expressions for lists.

3.2 Inter-Procedural Analysis

Once the source functions and the sink functions are determined, we perform a backward taint analysis. This analysis begins with the sink functions, which are a set of high-level printing or assembly functions located at the end of the output data flow. We then analyze the data flow to propagate taint backward to the determined source functions, as shown in Algorithm 1.

To better explain the program, we use an Interprocedural Control-Flow Graph (ICFG) instead of the Control-Flow Graph (CFG). Unlike the CFG, the ICFG contains two additional edges: call and return edges. The call edge represents the control flow from the caller to the callee, while the return edge indicates the reverse flow. By traversing the ICFG, we analyze the calling and called relationship for each function that is called. To ensure a context-sensitive data flow analysis, we use a function memory map to keep track of the data-flow context and status. For each function and its corresponding state, we create a node in our data flow. We limit our analysis to data flows that are feasible for both sources and sinks. For each feasible edge between the source and sink functions, we calculate, update, and propagate the data-flow summary to all callers.

We record the input state of each basic block within the given function and calculate its outgoing state based on the following three different situations:

If the basic block contains a call instruction, we cache the context switching behaviors to accelerate the calculation in the `analyzeCallInContext` function. This caching mechanism helps improve the performance of the analysis when encountering call instructions within the program. In the `getSummary` function, we reuse the existing analysis results if the summary is outdated and the new input state matches the old-n summary. However, if the new input state differs, we reanalyze this call with the updated summary input state and caller information.

If the basic block contains a Phi node, which is a special instruction in the LLVM framework used

Algorithm 1: Inter-Procedural Analysis

```
Function analysisBasicBlock($f$,$c$):
    /* $f$: the analysis target function */
    /* $c$: the input state */
    forall BasicBlock $b \in f$ do
        if $b$ contains outdated call then
            $c$.update()
            analysisFunction($b$.Callee, $c$)
        else
            if $b$ contains Phi node then
                forall Incoming BasicBlock $ib$ do
                    $c$.update($ib$.Context)
                end
            else if $f$ is source then $c \leftarrow$ taint($b$)
            propagate($b$)
        end
    end
end

/* Entry of the inter-Procedural analysis, */
/* analyze the whole program */
Function analysisInterProcdural($a$):
    /* $a$: the analysis target program */
    Context $c \leftarrow \emptyset$
    forall Function $f \in a$ do
        analysisFunction($f$,$c$)
    end
end
```

for merging incoming values from different predecessor basic blocks, we perform a Phi resolution. This involves calculating the output state individually for each possible incoming value with the given input state. We then assign the computed output states to the corresponding following basic blocks in the `meetOverPHI` function. This resolution step ensures that the appropriate output state is propagated based on the different incoming values.

If the basic block is within a source function, the transfer function determines if the basic block belongs to a source function and initiates the propagation process. The `transfer` function plays a crucial role in propagating the data flow within the source function, ensuring that the relevant data and state information are properly analyzed and propagated. By considering these different situations and applying the respective functions, we are able to accurately track the data flow and determine the outgoing state for each basic block within the given function.

3.3 Intra-Procedural Analysis

Based on the context-sensitive inter-procedural analysis, an intra-procedural analysis described in Algorithm 2 is performed for each derived function.

Within the basic block calling sinks, we extract output context with a type-specific sink extractor and build it into a production rule. If only one value is extracted from the sink, we append a terminal token to the basic block production. Otherwise, we append the conjunction of all possible values to the basic block production. For each basic block, we build a control flow graph and calculate the constraints with a checker, `PathChecker`.

Algorithm 2: Intra-Procedural Analysis

```
/* Analysis Path Context-Sensitively */
Function analysis($p$,$c$):
    if $isBranch(p)$ then
        $DestMap \leftarrow \emptyset$
        split ($p$)
        // Split $p$ into $(p_0,\ldots,p_i)$,
        // where $p_0$ is the default path
        for $\forall p_i, p_j \in p$, and $p_i.Dest = p_j.Dest$ do
            if $p_i == p_0$ then delete $p_j$
            else Disjunction ($p_i.c$, $p_j.c$)
        end
        for $\forall p_i.c \notin DestMap$ do
            $DestMap$.add ($p_i.c$)
            analysis ($p_i$, $c.case_i$)
        end
    end
end

/* Entry of the intra-procedural analysis, */
/* analyze the whole function */
Function analysisFunction($f$):
    /* $f$: the analysis target function */
    $b \leftarrow f.firstBasicBlock$    // BasicBlock
    $p \leftarrow b.start$                  // Path
    $c \leftarrow \emptyset$                    // Condition
    analyze ($p$, $c$)
end
```

For branch analysis, since there may be multiple cases (including the default one) going to the same destination, the path constraints are aggregated into a disjunction as in Algorithm 2 from line 2 to line 12. First, a destination map collects path conditions that lead to the same destination. If any of the possible cases share the same destination, a condition disjunction is created or updated (line 7). We use a cache to store newly created path conditions. Whenever a cache miss happens, we need to construct a fresh value. Especially, if any two cases share the same destination with the default, the other case would be removed (line 6), because the default path condition guarder can be obtained by negating all other conditions.

4 Evaluation

In this section, we evaluate our tool on real-world applications by extracting their output language grammar and generate output samples.

4.1 Experiment Setup and Subjects

For the sake of independence, constancy and reproducibility, we conduct all experiments within docker containers living in a dedicated host machine. The configuration of the host machine and docker containers is shown as follows.

- CPU: Intel Xeon E5–2690
- Memory: 378 GB
- OS: Ubuntu 18.04 LTS
- Docker Base Imagine: ubuntu:16.04
- Compiler: GCC 5.4.0, Clang 6.0.1
- Linker: GNU gold linker 1.11

We evaluate PRETTYGRAMMAR based on several small test cases collected from GitHub. We build these projects with clang as well as GNU gold linker and archive all the temporary files.

Table 1: Subjects revolved in evaluation.

Target Language	Output Program	Input Program
Static String	HelloWorld, staticXMLPrinter	
Dynamic String	loopPrinter, XMLPrinterClass	
Expression	printExpression	
Algebraic Equation	aePrinter	
XML	testWriter	xmllint

The related subjects are listed in Table 1. TestWriter is a test program that comes with the libxml2 project. Xmllint is a tool within the libxml2 project that parses XML files and outputs the result of the parsing.

4.2 Correctness of Inferred Grammar

To analyze the accuracy of the output grammar abstracted by PRETTYGRAMMAR, we collected three small programs that produce human-understandable output language: a static string printer (HelloWorld), an algebraic equation printer (aePrinter), several xml printers (staticXMLPrinter, XMLPrinterClass), and an expression printer (printExpression).

First, we evaluate PRETTYGRAMMAR on several static printers and successfully get its precise output grammar. The test indicates that PRETTYGRAMMAR is able to identify all source and sink functions of different types of terminal as designed. The exact value of printed terminals are also abstracted correctly.

Next, we evaluate PRETTYGRAMMAR on two printers that requires input and contains conditional output branches. The raw result for each program is shown in Figure 1. Figure 1a shows that PRETTYGRAMMAR is able to inferring the type of terminals, and interpreting production guards. Since we have performed context-sensitive and path-sensitive taint analysis, PRETTYGRAMMAR abstracts three different non-terminals from the same function printer::printExpression(), each from a different set of input and output variable status. However, as we can see in Figure 1a, some deterministic terminals such as `"="` and `"\n"` have been identified as possible terminals. This is a common false positive caused by the implementation of security check in the llvm and C++ printing methods. It can be eliminated by identifying the characteristic data-flow and control-flow pattern.

4.3 Performance on Real-World Applications

An ideal program language analysis tool should be able to assist researchers effectively, correctly, and efficiently. In this part, we evaluate PRETTYGRAMMAR from three aspects. We first generate corpus with the algorithm described in the following part, and then feed the corpus to a program that takes valid input under the same language with the output we abstracted.

4.3.1 Effectiveness

For software testing, the coverage of input samples are crucial. A high coverage sample are much more meaningful than vast random samples which are likely rejected in a very early stage of the program process. Thus, we observe the average coverage of source code in terms of lines and functions of our 1000 generated corpus, and list the results in Table 2,

```
<Start> ::= <main@0@0>
<main@0@0> ::= <print::printout()@0@d7a7a0>
<print::printout()@0@d7a7a0> ::= ("Illegal␣operation." | "<!dec!>" "<!string!>"? "<!dec!>" "="? "<!
    dec!>") "\n"?
```

(a) Algebraic equation printer

```
<Start> ::= <main@0@0>
<main@0@0> ::= <printer::printExpression(Exp*, bool)@0@efc0b0>
<printer::printExpression(Exp*, bool)@0@efc0b0> ::= "("? "<!string!>"? "<!string!>"? ("<!string!>" |
    <printer::printExpression(Exp*, bool)@efc0b0@ef9350> | "") ")"? "\n"?
<printer::printExpression(Exp*, bool)@efc0b0@ef9350> ::= "("? "<!string!>"? "<!string!>"? ("<!string
    !>" | <printer::printExpression(Exp*, bool)@ef9350@ef9350> | "") ")"? "\n"?
<printer::printExpression(Exp*, bool)@ef9350@ef9350> ::= "("? "<!string!>"? "<!string!>"? ("<!string
    !>" | <printer::printExpression(Exp*, bool)@ef9350@ef9350> | "") ")"? "\n"?
```

(b) Expression printer

Figure 1: The abstracted context-free grammar of simple printers

Table 2: Corpus average static coverage of xmllint.

Corpus	Coverage #		Coverage Ratio				
	Line (Out of 72998)	Function (Out of 3122)	Line	Function	Branch (Out of 1383)	Branch Taken at least Once	Call (Out of 763)
empty	1819	179	2.5%	5.8%	11.42%	5.86%	1.57%
random strings	1845.1	180.1	2.5%	5.8%	11.42%	5.86%	1.57%
PRETTYGRAMMAR	**2858.0**	**249.8**	**3.9%**	**8.0%**	**16.34%**	**8.39%**	**2.10%**
valid	2765.4	243.3	3.8%	7.8%	**16.34%**	**8.39%**	**2.10%**

where we refer the code coverage of one static input as static coverage, knowing form the dynamic generate or mutated input during the fuzz testings, in which we refer the coverage as dynamic coverage. Due to the scalability of `gcov`, which is the coverage analysis tool we adopt, we can only count the coverage of libxml2.

As shown in Table 2, corpus generated by PRETTYGRAMMAR achieves the best performance in static coverage. PRETTYGRAMMAR has outperformed empty and random samples in all coverage indicators. The result of random strings almost evens with the empty one, indicating that in such situations, random input is very unlikely to pass the basic input checker as well as the empty input in real-world applications. However, corpus generated by PRETTYGRAMMAR has reached source code in a decent level. In terms of software testing, trigging more source code can greatly increase the chance of finding new bugs or trigging new crashes. Compared with empty and random corpus, PRETTYGRAMMAR generated corpus has greatly improved the effectiveness of corpus coverage.

On the other hand, PRETTYGRAMMAR also outperforms valid corpus which are grammar-correct xml files we collected. We consider two possible reasons: 1) PRETTYGRAMMAR generates some incorrect samples and triggered error handling path that the valid samples would never touch. 2) After observing some samples form both corpus, we notice both the structure complexity and the length of synthesis samples is apparently higher than collected ones. While larger and more complex inputs are more likely triggering more source code.

Table 3: Correctness of XML, Checked by libxml2 xmllint

Check Result	Fraction (%)
Semantic Correct	**3.22**
Syntactic Correct	**38.7**
Syntactic Error	**58.06**

4.3.2 Correctness

To evaluate the correctness of the synthesized grammar, we performed an evaluation targeting the libxml2 XML linter `xmllint`. We fed the corpus we generated from libxml2 to `xmllint` and collected the feedback in the Table 3.

Table 3 presents the correctness evaluation results of the synthesized grammar for XML checked by libxml2 xmllint. The table shows the fraction of the check results in three categories: semantic correct, syntactic correct, and syntactic error. Among the tested corpus, only 3.22% are semantically correct, while 38.7% are syntactically correct, and 58.06% are syntactically incorrect.

5 Future Work

While our approach shows promising results, there are several directions for future work that could improve the algorithm's effectiveness and applicability. We discuss some of these potential directions below.

5.1 Handling More Complex Languages

Our current algorithm can handle languages with both lexical and syntactical structures, as well as entity identifiers. However, it may struggle with more complex languages that include features such as nested structures or complex type systems. One direction for future work could be to extend the algorithm to handle more complex languages, potentially by handle more complex languages.

5.2 Improving Precision of Static Analysis

Our static analysis method currently can partially infers possible values for string variables at each program point. While this is useful for generating valid input strings, it may not capture all possible behaviors of the program. In future work, we could explore more advanced static analysis techniques to improve the precision of our inferred string values, potentially by leveraging more advanced static analysis techniques.

5.3 Applying the Algorithm to Real-World Programs

While we demonstrate the effectiveness of our approach on a set of small example programs, it remains to be seen how well it would perform on more real-world programs. Future work could involve applying our algorithm to a wider range of programs and evaluating its effectiveness in generating high-quality input strings. Additionally, we could explore the feasibility of integrating our approach into existing software testing frameworks.

5.4 Integration with Fuzzing Techniques

Our approach provides a useful tool for generating input strings for software testing, but it does not directly address the process of actually testing the software. Future work could involve integrating our algorithm with existing fuzzing techniques to automatically generate and test input strings. This could potentially involve leveraging machine learning techniques to guide the generation of input strings towards unexplored parts of the program.

6 Conclusion

Deriving source code implied language is significant for a wide variety of applications. In this paper, we propose a static analysis that learns the implicit language from a program's source code. Our approach performs context-sensitive and path-sensitive taint analysis within the targeted class. To maintain context-sensitivity, we assign indirect calls with a potential callee pool and propagate the context environment to every possible candidate in the pool. To maintain path sensitivity, we represent conditional branches as nodes with constraints.

We implemented a prototype called PRETTYGRAMMAR in C++, based on the LLVM framework. Our experiments demonstrate that PRETTYGRAMMAR is effective and efficient in extracting grammar structures and generating output corpora for desired program output languages, such as XML.

Furthermore, we evaluated PRETTYGRAMMAR's output grammar using libxml2's XML linter, xmllint, and found that a large proportion of generated samples were syntactically correct.

Overall, our proposed approach shows promise in automatically generating program output corpora and can benefit a range of applications, including software testing, reverse engineering, and vulnerability analysis.

Acknowledgements

This research was supported in part by the National Science Foundation (NSF) grant CNS-1652790.

References

Adobe Systems Incorporated (2000). *PDF Reference*.

Ammons, G., Bodík, R., and Larus, J. R. (2002). Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16.

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., and Teuchert, D. (2019). Nautilus: Fishing for deep bugs with grammars. In *Network and Distributed System Security Symposium*, Reston, VA. Internet Society.

Chen, Y., Zhong, R., Hu, H., Zhang, H., Yang, Y., Wu, D., and Lee, W. (2021). One Engine to Fuzz'em All: Generic

Language Processor Testing with Semantic Validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (IEEE S&P)*, pages 1–17.

Coppit, D. and Lian, J. (2005). Yagg: an easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, page 356, New York, New York, USA. ACM Press.

for Standardization (ISO), I. O. (2008). ISO 32000-1: 2008 Document Management–Portable Document Format–Part 1: PDF 1.7.

Godefroid, P., Kiezun, A., and Levin, M. Y. (2008). Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA. IEEE Press.

Gopinath, R., Mathis, B., and Zeller, A. (2020). Mining input grammars from dynamic control flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 172–183, New York, NY, USA. ACM.

Groß, S. (2018). Fuzzil: Coverage guided fuzzing for javascript engines. *Master's thesis, TU Braunschweig*.

Guo, R. (2017). MongoDB's JavaScript fuzzer. *Communications of the ACM*, 60(5):43–47.

Harkous, H., Groves, I., and Saffari, A. (2020). Have Your Text and Use It Too! End-to-End Neural Data-to-Text Generation with Semantic Fidelity. *COLING 2020 - 28th International Conference on Computational Linguistics, Proceedings of the Conference*, pages 2410–2424.

Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA. USENIX Association.

Höschele, M. and Zeller, A. (2016). Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 720–725, New York, NY, USA. ACM.

Lin, Z. and Zhang, X. (2008). Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, page 83, New York, New York, USA. ACM Press.

Lin, Z., Zhang, X., and Xu, D. (2010). Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering*, 36(5):688–703.

Maurer, P. M. (1990). Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55.

Ruderman, J. (2007). Introducing jsfunfuzz. *URL http://www. squarefree. com/2007/08/02/introducing-jsfunfuzz*, 20:25–29.

Sirer, E. G. and Bershad, B. N. (1999). Using production grammars in software testing. *ACM SIGPLAN Notices*, 35(1):1–13.

Tillmann, N. and de Halleux, J. (2008). Pex–White Box Test Generation for .NET. In Beckert, B. and Hähnle, R., editors, *Tests and Proofs*, pages 134–153, Berlin, Heidelberg. Springer Berlin Heidelberg.

Toffola, L. D., Staicu, C., and Pradel, M. (2017). Saying 'Hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49.

Valotta, R. (2012). Taking browsers fuzzing to the next (dom) level. *Proceedings of the DeepSec*.

Veggalam, S., Rawat, S., Haller, I., and Bos, H. (2016). IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In Askoxylakis, I., Ioannidis, S., Katsikas, S., and Meadows, C., editors, *Computer Security – ESORICS 2016*, pages 581–601, Cham. Springer International Publishing.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2017). Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 579–594. IEEE.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2019). Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, volume 2019-May, pages 724–735. IEEE.

Wu, Z., Johnson, E., Yang, W., Bastani, O., Song, D., Peng, J., and Xie, T. (2019). REINAM: reinforcement learning for input-grammar inference. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 488–498, New York, New York, USA. ACM Press.

Xie, T., Tillmann, N., de Halleux, J., and Schulte, W. (2009). Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems Networks*, pages 359–368.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation - PLDI '11*, page 283, New York, New York, USA. ACM Press.

Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., and Wu, D. (2020). SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, New York, NY, USA. ACM.