AsFuzzer: Differential Testing of Assemblers with Error-Driven Grammar Inference

Hyungseok Kim
The Affiliated Institute of ETRI
Daejeon, South Korea
hskim@nsr.re.kr

Soomin Kim
KAIST
Daejeon, South Korea
soomink@kaist.ac.kr

Jungwoo Lee
KAIST
Daejeon, South Korea
jwlee2217@kaist.ac.kr

Sang Kil Cha
KAIST
Daejeon, South Korea
sangkilc@kaist.ac.kr

Abstract

Assembler is a critical component of the compiler toolchain, which has been less tested than the other components. Unfortunately, current grammar-based fuzzing techniques suffer from several challenges when testing assemblers. First, each different assembler accepts different grammar rules and syntaxes, and there are no existing assembly grammar specifications. Second, not every assembler is open-source, which makes it difficult to extract grammar rules from the source code. While existing black-box grammar inference approaches are applicable to such closed-source assemblers, they suffer from the scalability issue, which renders them impractical for testing assemblers. To address these challenges, we propose a novel way to test assemblers by automatically inferring their grammar rules with only a few queries to the target assemblers by leveraging their error messages. The key insight is that assembly error messages often deliver useful information to infer the underlying grammar rules. We have implemented our technique in a tool named AsFuzzer, and evaluated it on 4 real-world assemblers including Clang-integrated assembler (Clang), GNU assembler (GAS), Intel's assembler (ICC), and Microsoft macro assembler (MASM). With AsFuzzer, we have successfully found 497 buggy instruction opcodes for six popular architectures, and reported them to the developers.

CCS Concepts

• **Software and its engineering → Software testing and debugging.**

Keywords

assembler testing, grammar inference, compiler testing

ACM Reference Format:
Hyungseok Kim, Soomin Kim, Jungwoo Lee, and Sang Kil Cha. 2024. AsFuzzer: Differential Testing of Assemblers with Error-Driven Grammar Inference. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680345

1 Introduction

Compilers are crucial tools for building software because every binary running on a computer should have been processed by it at some point. Therefore, compiler correctness is a fundamental concern for software engineering. Assembler, a critical component of the compiler toolchain, is not an exception. If an assembler is buggy, the resulting program may not work as expected.

There have been extensive efforts to test the correctness of compilers [12], but most existing approaches target the whole compiler toolchain, which includes an assembler, by generating test cases in a higher-level language, such as C [59] and CIL [17]. Such a holistic approach is insufficient to achieve high code coverage as only a number of assembly instruction mnemonics known by the code generator can be tested. Furthermore, an assembler component may not be tested at all if the compiler does not produce intermediary assembly code during the machine code emission process. For example, LLVM writes object files without an assembler unless inline assembly is used [35]. Thus, it is imperative to *individually* test an assembler with a variety of assembly instructions in order to discover potential bugs in it.

Grammar-based fuzzing [14, 19, 23, 39, 40, 44] is a promising approach to testing assemblers, but there are several remaining challenges. First, each assembler implementation accepts different assembly grammar rules and syntaxes even for the same CPU architecture. For example, Intel AT&T syntax from GNU assembler has different operand ordering than the Intel ISA manual. Second, not every assembler is open-sourced, making it difficult to obtain the grammar of assembly instructions from the source code. For example, Microsoft macro assembler is the major assembler for Windows, but its source code is not publicly available.

One potential solution to these challenges is to infer the grammar of assembly instructions in a black-box manner, and then use the inferred grammar to generate assembly instructions. There are indeed several recent black-box approaches [7, 33] that only require a set of examples and an oracle to infer context-free grammars. The oracle returns either "yes" or "no" depending on whether a given string is valid under the target grammar or not. Therefore, one may regard our target assembler as an oracle and provide it with several assembly instructions as examples to infer the grammar rules of a closed-source assembler implementation.

Unfortunately, however, these black-box approaches suffer from a scalability issue. Their time complexity is known to be $O(n^4)$, where n is the total length of the input files [7] even with various heuristics. In our study, we observe that the existing approaches do *not* scale well for complex assembly languages such as x86-64, which has more than 1,000 opcodes and various addressing modes.

Therefore, we present a novel approach to *efficiently* infer the grammar of assembly instructions that a given assembler accepts. The key insight of our approach is that error messages generated by an assembler often provide useful information about the grammar of the underlying assembly language that the assembler accepts. Thus, we can gradually narrow down the search space of the grammar rules by leveraging the error messages. As an example, consider an `add` instruction in x86-64 assembly, which should always be followed by two operands. When we try to assemble an `add` instruction with four operands, e.g., "`add 1, 1, 1, 1`", GNU Assembler (GAS) will produce an error message saying that the number of operands is incorrect. When we modify the instruction to have three operands, e.g., "`add 1, 1, 1`", we observe the same error, but if we make it to have two operands, e.g., "`add 1, 1`", GAS emits a different error message saying that the types of operands are incorrect. With these observations, we can infer that the `add` instruction does not accept four/three operands, but two, thereby reducing the search space of the grammar rules. This simple idea enables us to infer the grammar rules of an instruction within a constant number of assembler queries.

With the inferred assembler-specific grammar, we can then perform grammar-aware differential testing on the target assemblers. Suppose we have obtained grammars for two different assemblers targeting x86-64. We first select a random opcode in x86-64 and check what kind of syntaxes each assembler accepts for the opcode. For a syntax that is accepted by the two assemblers, we randomly generate assembly instructions following the syntax, assemble them with each assembler, and compare the outputs to detect potential bugs as in traditional differential testing [31, 46, 48, 51]. For a syntax that is accepted by only one of the assemblers, we *cannot* simply use the differential testing approach because the other assembler will reject assembly instructions following the syntax. Furthermore, we cannot simply deem one of the assemblers to be buggy because it is possible that the other assembler may not simply support the particular syntax. Thus, we devise a way to detect assembler-specific bugs by leveraging a disassembler as a reference implementation. In particular, we assemble the generated assembly instructions with the target assembler and disassemble the generated machine code with the reference disassembler. We then compare the disassembled instructions with the original assembly instructions to detect potential bugs in the target assembler.

We implemented our idea in a tool named AsFuzzer and evaluated it with four mainstream assemblers: Clang-integrated assembler (Clang), GNU assembler (GAS), Microsoft macro assembler (MASM), and Intel's assembler (ICC). The results are promising. We found a total of 497 previously unknown bugs, identified by unique instruction opcodes, from the four assemblers (142, 61, 210, and 84 bugs from Clang, GAS, ICC, and MASM, respectively) and reported them to the developers. Our contributions are as follows.

(1) We propose a novel approach to efficiently infer the grammar of assembly instructions that a given assembler accepts by leveraging error messages generated by the assembler.
(2) We propose a novel differential testing approach to detect potential bugs of an assembler by leveraging the inferred grammar and a disassembler as a reference implementation.
(3) We implemented our approach in a tool named AsFuzzer and evaluated it with four real-world assemblers including closed-source ones.
(4) We publicize our tool to facilitate future research on assembler testing: https://github.com/SoftSec-KAIST/AsFuzzer.

Table 1: Comparison of recent testing tools that generate assembly code.

Target	Tool	Year	Test Case Language	Tool-Specific Grammar	x86	x86-64	ARM	AArch64	MIPS	RISCV64
Compiler	DeepFuzz [42]	2019	C	✗	-	-	-	-	-	-
	YARPGen [43]	2020	C/C++	✗	-	-	-	-	-	-
	CompDiff [41]	2023	C/C++	✗	-	-	-	-	-	-
CPU	DifuzzRTL [23]	2021	Assembly	✗	✗	✗	✗	✗	✗	✓
	TheHuzz [29]	2022	Assembly	✗	✗	✗	✗	✗	✗	✓
Assembler	AsFuzzer (ours)	-	Assembly	✓	✓	✓	✓	✓	✓	✓

2 Background and Motivation

This section discusses relevant research on generating assembly code as a test case, and motivates our approach by presenting an example bug found by AsFuzzer.

2.1 Generating Assembly Code

Compiler testing has been an active research area for decades [11–13, 37, 38, 41–43, 49, 52, 55, 56, 59, 60]. Despite burgeoning research in this area, most existing approaches overlook the assembler component of compilers. While it is possible to generate valid assembly code by compiling random C programs, e.g., with programs generated by Csmith [59], we cannot achieve high code coverage with a limited set of assembly instructions known by the compiler.

There are several recent attempts in synthesizing assembly instructions to test RISCV64 processor implementations [23, 29], although their testing targets are neither compilers nor assemblers. They use a manually written set of grammar rules to generate valid assembly instructions. However, their focus is on making generally valid assembly instructions, and do not consider the syntaxes of assembly instructions specific to each assembler.

Table 1 summarizes the relevant tools appeared in top-tier venues in the past five years (2018–2023). The first column indicates the testing target of each tool. The second and third columns present the names of the tools and their publication years, respectively. The fourth column shows in which language their test cases are written, and the fifth column describes whether they use an assembler-specific grammar to generate test cases. The rest of the columns show which CPU architecture each tool can handle. From the table, we can clearly see that our tool is the first in targeting assemblers. Moreover, our approach is scalable in that it can handle a wide range of CPU architectures, which is made possible by our novel approach to automatically inferring assembly grammars.

```
vcvtusi2ss XMM0, XMM0, RAX, 0xc3
```

(a) An assembly instruction generated by AsFuzzer.

```
62 f1 fe 08 7b c0        vcvtusi2ss XMM0,XMM0,RAX
c3                       ret
```

(b) Machine instructions obtained by compiling the assembly code in (a) with GAS.

Figure 1: Example bug in GAS found by AsFuzzer.

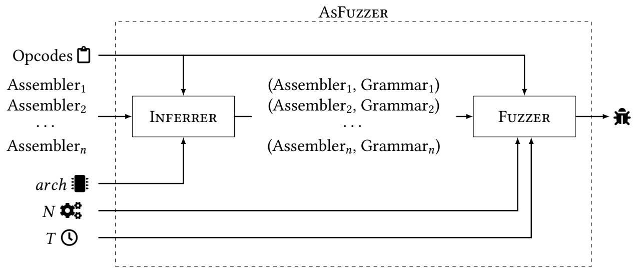

AsFuzzer
Opcodes
Assembler$_1$
Assembler$_2$
...
Assembler$_n$
arch
N
T
Inferrer
(Assembler$_1$, Grammar$_1$)
(Assembler$_2$, Grammar$_2$)
...
(Assembler$_n$, Grammar$_n$)
Fuzzer

Figure 2: AsFuzzer architecture.

2.2 Motivating Example

We now present an example to motivate our approach. Figure 1 shows a previously unknown bug found by AsFuzzer in GNU assembler (GAS). By feeding in the assembly instruction shown in Figure 1a as input to GAS, it produces two instructions shown in Figure 1b: `vcvtusi2ss` followed by `ret`. Note that the instruction `vcvtusi2ss` in our example *unusually* has four operands, while the Intel manual [24] states that it should have only three operands. That is, GAS has a bug where it accepts this wrong assembly instruction and produces two machine instructions as a result.

The compiler testing tools shown in the top three rows of Table 1 as well as traditional ones like Csmith [59] are not effective in finding this bug because it is extremely unlikely to generate such a wrong assembly instruction by compiling a regular C program. Moreover, it is extremely unlikely for a compiler to emit such an esoteric opcode (regardless of the number of operands) unless the given C program uses an intrinsic function, e.g., `_mm_cvt_roundu32_ss` in ICC, that directly maps to the `vcvtusi2ss` instruction.

The CPU fuzzing tools shown in the fourth and fifth rows of Table 1 will also not be effective in finding this bug even if they can generate x86-64 assembly instructions as they only consider generating *valid* instructions that follow the general x86-64 grammar. Unless the grammar includes the buggy syntax, they will never be able to generate such a wrong assembly instruction. Furthermore, since they are dependent on manually written grammar rules, it is difficult to apply them to different assemblers that accept different assembly syntaxes or target different CPU architectures.

On the other hand, AsFuzzer can effectively find this bug by inferring the GAS-specific syntaxes of assembly instructions. In particular, AsFuzzer will identify that GAS accepts two forms of `vcvtusi2ss`: one with three operands and another with four operands. It will then realize that the latter form is never accepted by other assemblers. Thus, it will generate a random assembly instruction i with four operands following the inferred syntax, and assemble it with GAS. GAS will then produce the two machine instructions as shown in Figure 1b. Next, AsFuzzer will disassemble the two instructions and notice that the disassembled instructions are different from i, and hence, it will report this instance as a bug.

3 AsFuzzer Design

We now describe the design of AsFuzzer. We first give a brief overview of AsFuzzer, and then describe the two main modules of AsFuzzer: Inferrer and Fuzzer.

3.1 Overview

AsFuzzer consists of two main modules: Inferrer and Fuzzer as depicted in Figure 2. At a high level, AsFuzzer takes in five arguments as input: (1) a CPU architecture *arch*, (2) a list of assemblers supporting the CPU architecture, (3) a list of available opcodes for the architecture, (4) the number of fuzz iterations per opcode N, and (5) the timeout of a fuzzing campaign T. AsFuzzer then returns a set of bugs found from the assemblers as output.

Inferrer: `Arch × Assembler [] × Opcode [] → (Assembler × Grammar) []`
Inferrer takes in as input (1) a CPU architecture *arch*, (2) a list of assemblers that support *arch*, and (3) a list of available opcodes for *arch*. One can easily obtain the list of available opcodes for *arch* from the architecture reference manual [6, 24], or from the source code of the existing assemblers. Inferrer iterates every given opcode for each assembler, infers operand formats for every opcode (denoted as `Grammar`), and makes a pair of each assembler and its inferred grammar. Finally, it returns a list of such pairs for all assemblers. We further detail the algorithm of Inferrer in §3.2.

Fuzzer: `(Assembler × Grammar) [] × Opcode [] × Iter × Timeout → Bug []`
Fuzzer takes in as input (1) a list of (assembler, grammar) pairs obtained from the inferrer module, (2) a list of available opcodes, (3) the number of iterations N to fuzz per opcode, and (4) the timeout T. It then iteratively performs differential testing to find bugs in the given assemblers. First, it selects an opcode at random. It then randomly selects an assembler A that can consume the selected opcode, and picks a random operand format that A can accept for the opcode. Next, it generates a random instruction i for each of the selected rules, and assembles it using the selected assembler A to produce a binary b_A. It then checks for all the other assemblers if the same operand format exists for the opcode. For an assembler B that has the same format, we assemble i with B to obtain a binary b_B, and compare b_B with b_A to see if there is any difference. For an assembler C that does not have the same format, we cannot assemble i as C will reject it. Instead, we disassemble b_A with a reference disassembler, e.g., GNU `objdump`, to obtain i', and compare i with i' to see if there is any difference. We repeat the whole process until the timeout T is reached while producing N random instructions per each iteration. We further detail the algorithm of Fuzzer in §3.3.

Algorithm 1: Inferrer.

```
1  function Inferrer ($arch, assemblers, opcodes$)
2      $pairs \leftarrow [\cdot]$ // empty list
3      $fmts \leftarrow$ get_all_possible_operand_formats($arch$)
4      for $asm \in assemblers$ do
5          $grammar \leftarrow \{\cdot\}$ // empty dictionary
6          for $op \in opcodes$ do
7              $grammar[op] \leftarrow \emptyset$
8              $cnt \leftarrow$ num_operands($arch, asm, op$)      // §3.2.1
9              for $c \in cnt$ do
10                 $f \leftarrow$ infer($arch, asm, op, fmts, c$)   // §3.2.2
11                 $f' \leftarrow$ filter($asm, f$)               // §3.2.3
12                 $grammar[op] \leftarrow grammar[op] \cup f'$
           // append to the list
13         $pairs \leftarrow pairs + (asm, grammar)$
14     return $pairs$
```

3.2 Inferrer Module

Algorithm 1 shows the pseudocode of Inferrer. In Line 2, we first initialize a list that will contain pairs of an assembler and its inferred grammar. In Line 3, we obtain all possible operand formats for the given architecture *arch*. An operand format is a nonterminal symbol in the grammar that represents a possible operand syntax for an opcode. AsFuzzer has a predefined set of operand formats for each architecture, and `get_all_possible_operand_formats(`*arch*`)` returns such a set for *arch* as listed in Table 2.

The algorithm then iterates every given assembler to infer available operand formats for every opcode. In Line 5, we prepare an empty dictionary *grammar*, which maps an opcode to a set of available operand formats for the opcode. In Line 7, we initialize the set of available operand formats for the opcode to be empty. In Line 8, we infer the number of operands that the opcode can take using our error-driven approach described in §3.2.1. At a high level, we leverage the fact that assemblers emit a specific type of error message when the number of operands is incorrect in order to efficiently infer available operand counts for an opcode.

Once we obtain the number of operands, we then derive valid operand formats for every operand count in Line 10. Note that there can be multiple valid operand formats for a given operand count. For example, when the number of operands is two, the opcode may accept two distinct operand formats: "reg64, reg64" and "reg64, imm", where "reg64" and "imm" mean a 64-bit register and an immediate, respectively. To derive all possible operand formats for a given operand count, we again leverage assembler error messages to infer valid operand formats as we further describe in §3.2.2.

Finally, in Line 11, we filter out the inferred operand formats that are specific to the given assembler. For example, some assemblers may accept a pseudo instruction that is not officially supported by the architecture. Thus, we identify and exclude such cases from *grammar* to reduce false positives in our analysis. We further detail this technique in §3.2.3. The final output of Inferrer is a list of pairs of an assembler and its inferred grammar.

3.2.1 *Error-Driven Operand Count Inference.*

We automatically infer the number of operands that an opcode can take using assembler error messages. The key insight is that assemblers often emit a specific type of error message when the number of operands is incorrect. For example, when we provide an `add` instruction of x86-64 with a wrong number of operands to GNU assembler, it emits the following error message: "`Error: number of operands mismatch for `add'`". We find that all the assemblers we tested except Clang emit a unique error message when the number of operands is invalid for an opcode. We name this type of error as *operand count error*.

By observing operand count errors, we can efficiently infer the number of operands that an opcode can take. Assuming that the maximum number of operands that the architecture supports is n, we simply generate $n + 1$ instructions with varying numbers of operands from 0 (no operand) to n where each operand is a randomly chosen register in the architecture *arch*. For example, we create five dummy instructions for the `add` opcode of x86-64: "`add`", "`add RAX`", "`add RAX, RAX`", "`add RAX, RAX, RAX`", "`add RAX, RAX, RAX, RAX`". We then check if the assembler emits an operand count error for each of the instructions. For those that do not emit an operand count error (while another type of error may still be emitted), we can infer that the opcode can take the number of operands that the instruction has. When our target assembler does not have a unique error message for an operand count error, as is the case for some assemblers like Clang, we simply return all possible counts from 0 to n, i.e., $\{0, 1, 2, 3, 4\}$ in our example. Such an over-approximation will *not* affect the precision of our analysis, although it will increase the inference time, because our format inference mechanism will filter out invalid operand formats in the next step anyway.

3.2.2 *Error-Driven Operand Format Inference.*

After obtaining possible operand counts for an opcode, we then infer valid operand formats for each count. For an `add` instruction on x86-64, for instance, we will get the possible operand count of two, which means that `add` instructions will always follow the instruction format: `add <op1>, <op2>`. Thus, the goal of this step is to infer valid operand syntaxes for each operand placeholder, i.e., `<op1>` and `<op2>`.

The simplest way to infer the possible operand formats is to try all possible combinations of operand values, i.e., all possible registers, immediate values, memory forms, and so on, for each operand placeholder and see if the assembler accepts it. However, there are too many such combinations to consider in practice.

To reduce the search space, we leverage the fact that assembler error messages are always similar when similar types of operands are used. Consider two invalid `add` instructions: "`add 1, RAX`", and "`add 2, RBX`". Both instructions are invalid because the first operand cannot be an immediate, and GNU assembler will emit the same error message for both instructions. Therefore, we do not need to try both 1 and 2 for the first operand, and similarly no need to try both `RAX` and `RBX` for the second operand, as they are under the same operand category.

To this end, we define a set of *operand types* for each architecture to group similar operand values together. We then try only a single instance for each operand type during the inference process. For Intel x86-64, for example, we have 37 predefined operand types

Table 2: Predefined operand types for x86-64. A total of 37 operand types are defined in AsFuzzer.

Type	Examples	Type	Examples
reg512	ZMM0, ...	mem_base	[RAX], ...
reg256	YMM0, ...	mem_base_disp	[RAX+1], ...
reg128	XMM0, ...	mem_disp	[1], [2], ...
reg80	ST0, ...	mem_base_zword	ZMMWORD PTR [RAX], ...
regmmx	MM0, ...	mem_base_disp_zword	ZMMWORD PTR [RAX+1], ...
reg64	RAX, ...	mem_disp_zword	ZMMWORD PTR [1], ...
reg32	EAX, ...	mem_base_yword	YMMWORD PTR [RAX], ...
reg16	AX, ...	mem_base_disp_yword	YMMWORD PTR [RAX+1], ...
reg8	AL, ...	mem_disp_yword	YMMWORD PTR [1], ...
imm	1, 2, ...	(18 more memory operands)	...

as shown in Table 2. To infer valid operand formats for the add instruction, we consider all combination of operand types for each operand placeholder: "reg512, reg512", "reg512, reg256", "reg512, reg128", and so on. For each combination, we select one operand instance for each operand type to make a concrete instruction and to check its validity. This means we need to make $1,369$ ($= 37 \times 37$) add instructions to figure out the valid operand formats. Creating a file for each instruction is inefficient as we would have to invoke an assembler for each instruction. Instead, we create a single assembly file that contains all the instructions as we can easily identify which instruction in the file caused an error message by looking at the line number of the error message.

One exception is MASM, which does *not* emit per-line error messages when there are more than 100 errors in a single assembly file. This means we can put at most 100 instructions in a single assembly file for MASM. Therefore, we need to create at least 14 ($\approx 1,369/100$) assembly files to handle the add instruction of x86-64 MASM. As a result, MASM incurs significantly more overhead than other assemblers in the inference step as we will discuss in §4.2.

3.2.3 Pseudo Instruction Filtering. Our error-driven approach produces a large number of valid instruction syntaxes for each individual assembler. However, some of them are too specific to the given assembler, and thus should not be generally considered as a valid syntax. In particular, we found that assemblers may accept *pseudo instructions* that are often used to ease the development of assembly programs. However, pseudo instructions are not guaranteed to be supported by all assemblers, and hence can produce potential false positives in our analysis. For example, "abs" on MIPS is a pseudo instruction that is assembled into three regular instructions with GAS, but other assemblers do not support it.

To filter out such pseudo instructions, we leverage the fact that pseudo instructions are always translated into a sequence of real instructions (with distinct opcodes) by the assembler. Therefore, we can identify pseudo instructions by disassembling the binary produced by the assembler and checking if the disassembled instruction opcodes are different from the original opcode. In our current implementation, we use GNU objdump as our reference disassembler. Such a simple process filters out 6.44% of the inferred operand formats for all the architectures we tested: x86, x86-64, ARM, AArch64, MIPS, and RISCV64. We note that this filtering process can overly reduce the number of valid operand formats because our disassembler can be buggy. However, this will only reduce the number of bugs that we can find, and will *not* affect the precision of our analysis.

Algorithm 2: Fuzzer.

```
// pairs: a list of (assembler, grammar)
1  function Fuzzer (pairs, opcodes, N, T)
2      bugs ← ∅
3      disasm ← GNU objdump // our reference disassembler
4      while ¬is_timeout(T) do
5          op ← pick_random_opcode(opcodes)
6          assembler_1, g_1 ← pick_random_pair(op, pairs)
           // remove the pair from the list
7          pairs' ← pairs − (assembler_1, g_1)
8          assembler_2, g_2 ← pick_random_pair(op, pairs')
9          instrs_1 ← [·] // empty list
10         instrs_2 ← [·] // empty list
11         for i ← 1 to N do
12             fmt ← pick_random_format(g_1[op])
13             ins ← gen_assembly_instruction(op, fmt)
14             if fmt ∈ g_2[op] then instrs_1 ← instrs_1 + ins
15             else instrs_2 ← instrs_2 + ins
16         b_1 ← diff_two_asms(assembler_1, assembler_2, disasm,
            instrs_1)                                   // §3.3.1
17         b_2 ← diff_asm_disasm(assembler_1, disasm, instrs_2)
            // §3.3.2
18         bugs ← bugs ∪ b_1 ∪ b_2
19     return bugs
```

3.3 Fuzzer Module

Recall from §3.1, Fuzzer performs differential testing in two distinct ways depending on the availability of the common operand format between two target assemblers. Algorithm 2 shows the overall workflow of Fuzzer. In Line 2, we start by initializing *bugs*, which will contain all the bugs found during the fuzzing campaign. In Line 3, we initialize our reference disassembler to be GNU objdump. The fuzzing loop then iterates until the timeout T is reached.

In Line 5, we first pick an opcode from the given list of opcodes (*opcodes*) at random. In Line 6–8, we then randomly pick two different assemblers ($assembler_1$ and $assembler_2$), which can consume the opcode, along with their grammars (g_1 and g_2). In Line 9–10, we initialize two empty lists of assembly instructions ($instrs_1$ and $instrs_2$). Next, we fill in the lists by iterating the for-loop for N times. In Line 12, we randomly pick an operand format that $assembler_1$ can accept. In Line 13, we generate a random assembly instruction *ins* that has the opcode *op* and conforms to the selected operand format fmt. In Line 14, we check if $assembler_2$ also accepts the same operand format fmt for the opcode *op*. If so, we accumulate the instruction *ins* to the list $instrs_1$. If otherwise, we accumulate *ins* to the list $instrs_2$.

After generating N instructions, we first perform differential testing between $assembler_1$ and $assembler_2$ in Line 16 by putting all the instructions in $instrs_1$ into a single file, and comparing the binaries produced by assembling the file with $assembler_1$ and $assembler_2$. In theory, any difference between the two binaries should result in a bug, but there can be false positives in practice. Hence, we devise an effective technique to reduce false positives as we further describe in §3.3.1.

Next, in Line 17, we solely test $assembler_1$ by checking the consistency between the input and the output of it, because $assembler_2$ does not accept any of the instructions in $instrs_2$. In particular, we first assemble all the instructions in $instrs_2$ with $assembler_1$ to produce a binary. We then disassemble the binary with our our reference disassembler (i.e., GNU `objdump`), and compare the disassembled instructions with $instrs_2$ to see if there is any difference. We further detail this process in §3.3.2.

3.3.1 Differential Testing between Two Assemblers. When two assemblers share the same operand format for an opcode, we can directly run them with the same input instruction and compare their output binaries to detect any difference. However, we found that an instruction can be encoded in multiple ways, resulting in different binaries depending on the assembler implementation. For example, an `add` instruction of x86-64 can be encoded in several ways depending on the size of the immediate: both $\texttt{480501000000}_{16}$ and $\texttt{4883c001}_{16}$ are valid encodings of the instruction "`add rax, 1`". Such differences in the encoding can result in *false positives*.

To handle this issue, we first disassemble the output binaries of the two assemblers with a reference disassembler (GNU `objdump`), and compare the disassembled instructions. This way, we can eliminate false positives caused by the difference in the encoding of the same instruction. Of course, our reference disassembler may produce a wrong disassembly or fail to disassemble the binary, in which case we may observe false negatives. One potential way to reduce false negatives is to use multiple reference disassemblers, but in our evaluation, GNU `objdump` was sufficient to detect many practical bugs in real-world assemblers. Hence, we leave it as future work to further improve the precision of our analysis.

3.3.2 Consistency Checking between Input and Output of an Assembler. When two assemblers do not share the same operand format for an opcode, we cannot perform differential testing between them. One may say that two assemblers having different operand formats for the same opcode is already a bug, but it is possible that one assembler simply does not support the syntax yet, i.e., an unimplemented feature, or the other assembler supports a special syntax that is not officially defined in the architecture reference manual.

To reduce false positives in such cases while still being able to detect potential bugs, we check the consistency between the input and the output of the selected assembler. In particular, we first assemble instructions $instrs_2$ (i.e., input) with the selected assembler $assembler_1$, and disassemble the binary produced by the assembler (i.e., output) with a reference disassembler (GNU `objdump`). We then compare the disassembled instructions with the original instructions to see if there is any difference. With this idea, we can detect inconsistency bugs similar to the one in Figure 1.

3.4 Implementation

We implemented AsFuzzer with 3.1K SLoC of Python. To obtain the list of opcodes for each architecture, we manually inspected the source code of GNU Binutils and extracted the lists from it. We also manually defined the set of operand formats (Line 3 of Algorithm 1) for each architecture by carefully examining the architecture reference manuals. For x86, x86-64, ARM, AArch64, MIPS, and RISCV64 architectures, we used 37, 37, 9, 10, 6, and 6 distinct operand formats, respectively. We use GNU `objdump` v2.41 as our reference disassembler.

4 Evaluation

In this section, we evaluate AsFuzzer to answer the following research questions.

RQ1. How does Inferrer compare to SOTA black-box grammar inference and automata learning approaches? (§4.2)
RQ2. How does our test case generation method compare to alternative approaches? (§4.3)
RQ3. How effective is AsFuzzer in finding bugs in mainstream assemblers? (§4.4)
RQ4. How do the assembler bugs we found look like? (§4.5)

4.1 Experimental Setup

4.1.1 Our Benchmark. Our benchmark includes four popular assemblers: two public assemblers and two proprietary assemblers. For public assemblers, we selected Clang-integrated assembler v16.0.0 (Clang) and GNU assembler v2.41 (GAS), which are the default assemblers of Clang and GCC, respectively. In our experiments, we used them to assemble assembly programs for x86, x86-64, ARMv7, AArch64, MIPS, and RISCV64. We also chose two proprietary assemblers: Intel's assembler v2021.8.0 (ICC) and Microsoft macro assembler v14.37.32824.0 (MASM). We use those two assemblers to assemble assembly programs for x86 and x86-64.

4.1.2 Comparison Target.

Grammar Inference Tool. To compare the effectiveness of our grammar inference algorithm, we selected Arvada [33], which is a SOTA black-box grammar inference tool, as our comparison target. Additionally, we selected two active automata learning algorithms, namely L* [4] and TTT [25] to infer assembly grammars. Our comparison focused on evaluating the inference time and accuracy of the inferred grammars.

Assembler Testing Tool. We were not able to find any existing assembler testing tools for comparison. Instead, we selected three tools from compiler testing, CPU fuzzing, and grammar-based fuzzing approaches as our comparison targets. Specifically, we chose Csmith [59], which is a standard C compiler testing tool that generates random C programs. With Csmith, we generated random C source files first, and then compiled them with the `--save-temps` option to produce assembly files from C. We manually truncated the last assembly file to get exactly 1M assembly instructions. We also chose DifuzzRTL [23], which is a SOTA CPU fuzzing tool that can generate random RISCV assembly programs. Finally, we included Grammarinator [18], which can generate test cases by leveraging existing assembly grammars defined in ANTLR v4 [36]. We found two existing assembly grammars written in ANTLR for x86 [1] and RISCV [2], so we used them to run Grammarinator. Note that an end-to-end comparison is not feasible with these tools because their goals and usages are different from AsFuzzer. Therefore, we only compared the effectiveness of input generation of each tool in §4.3.

Table 3: Time taken (in seconds) to infer assembly grammars.

Assembler	**x86**				**x86-64**				**ARMv7**				**AArch64**				**MIPS**				**RISCV64**			
	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT
Clang	**3,749**	T.O.	T.O.	23,040	**5,416**	T.O.	T.O.	52,200	**314**	1,286	1,758	1,441	**190**	7,574	2,209	1,506	**66**	3,125	880	673	**43**	4,084	1,037	587
GAS	**903**	T.O.	11,220	3,319	**798**	T.O.	12,360	4,380	**25**	1,126	376	300	**74**	5,472	493	356	**11**	1,875	213	170	**15**	3,092	208	169
ICC	**1,476**	T.O.	T.O.	27,420	**700**	T.O.	T.O.	22,440	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MASM	**10,709**	T.O.	T.O.	66,840	**13,621**	T.O.	T.O.	69,060	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

T.O. means timeout.
The numbers in bold represent the best result per row.

Table 4: Rate between the number of valid instructions and the total number of instructions generated by each tool.

Assembler	**x86**				**x86-64**				**ARMv7**				**AArch64**				**MIPS**				**RISCV64**			
	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT	AsFuzzer	Arvada	L^*	TTT
Clang	100%	T.O.	T.O.	100%	100%	T.O.	T.O.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
GAS	100%	T.O.	100%	100%	100%	T.O.	100%	100%	100%	85.8%	100%	100%	100%	91.3%	100%	100%	100%	80.0%	100%	100%	100%	90.6%	100%	100%
ICC	100%	T.O.	T.O.	100%	100%	T.O.	T.O.	100%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
MASM	100%	T.O.	T.O.	100%	100%	T.O.	T.O.	100%	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

4.1.3 Our Environment. We ran our experiments on a desktop machine with 16 Intel i9-11900 cores and 128GB of RAM. We allocated a single CPU core for each running instance. We used Ubuntu 20.04 containers on Docker 24.0.7 to run all our experiments. To run MASM, which can only run on Windows, we used Wine v3.0.

4.2 Effectiveness of Inferrer

How effective is Inferrer in inferring the grammar rules of an assembler? To answer the question, we chose one black-box grammar inference tool, Arvada, and two active automata learning algorithms, L^* [4] and TTT [25], as our comparison targets.

Since Arvada requires a set of examples to operate, we first generated a set of random assembly instructions. Specifically, we used AsFuzzer to enumerate example assembly instructions, and then randomly selected one instruction per each opcode. That is, if an ISA has N distinct opcodes, we give N valid assembly instructions (one random instruction per each opcode) as input to Arvada. We also used our assemblers (see §4.1.1) as membership oracles.

Since both L^* [4] and TTT [25] require an equivalence oracle to operate, we employed a PAC (Probably Approximately Correct) oracle [57] to perform stochastic equivalence testing. Specifically, we implemented the oracle using the LearnLib [26] framework with the error parameter $\epsilon = 0.01$ and the confidence parameter $\delta = 0.01$. Note that simply considering all possible opcodes and operands as an alphabet results in a significantly large number of possible combinations, making it infeasible to find counterexamples. Thus, we infer the grammar for each opcode separately by randomly sampling possible operand combinations. For example, when targeting an x86-64 assembler, we provide the learning module with 37 concrete operands, i.e., one instance per each of the operand types shown in Table 2.

4.2.1 Grammar Inference Time. We ran AsFuzzer, Arvada, and two learning modules, L^* and TTT, for 24 hours to compare their time efficiency in inferring grammar rules. Table 3 shows how much time it took to infer the grammar of each assembler for each architecture. Overall, AsFuzzer was significantly faster than the others for all the configurations. AsFuzzer was 7.2× faster than TTT, and for those cases where Arvada and L^* were able to infer the grammar rules within 24 hours, AsFuzzer was 37.5× faster than Arvada and 9.7× faster than L^*. AsFuzzer spent more time for MASM than for the other assemblers. We believe this is because of two main reasons. First, we used Wine to run MASM on Linux as it requires Windows to operate. Wine incurs a significant API translation overhead, which can slow down the inference process. Additionally, AsFuzzer generated more assembly files for MASM, as discussed in §3.2.2, which also increased the inference time.

Note that Arvada and L^* were not able to infer the grammar of Intel assemblers within 24 hours (denoted as T.O.). We believe this is because of the complexity of the Intel assembly language, which has thousands of distinct opcodes and 3×–4× more operand types compared to the other architectures. These results signify the efficiency of our error-driven grammar inference algorithm.

4.2.2 Grammar Accuracy. To compare the accuracy of the inferred grammars, we performed two experiments. First, we compared the precision of the inferred grammars by AsFuzzer, Arvada, L^*, and TTT. Second, we compared the coverage of the inferred instruction formats by each tool. For the second experiment, however, we were

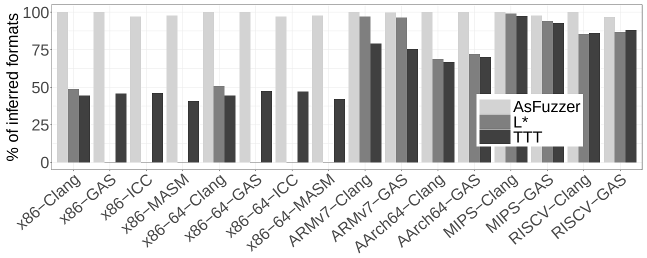


Figure 3: The percentages of instruction formats covered by AsFuzzer, L*, and TTT.

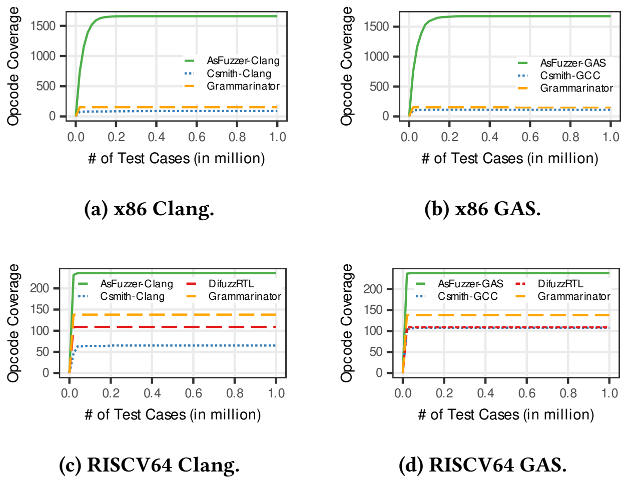
not able to use Arvada because it was not straightforward to modify the tool to generate acceptable assembly formats.

Grammar Precision. Since there is no ground truth, we first measured the precision, but not recall, of the inferred grammars. Specifically, we generated 10,000 assembly instructions from each tool, and then fed them to the oracle assemblers to see how many of them were accepted. Table 4 shows the results. AsFuzzer, L*, and TTT achieved 100% precision for all the assemblers, while Arvada achieved significantly less precision. We further analyzed the results and found that Arvada produced many invalid instructions with invalid opcodes or operands. For example, Arvada produced an invalid instruction "`yieldmovk X7, #20`" for AArch64, where `yield` and `movk` are two distinct opcodes. From the results, we conclude that AsFuzzer is more precise than Arvada in terms of inferring assembly grammar rules.

Grammar Coverage. To measure the grammar coverage of each tool, we enumerated all possible instruction formats that each tool inferred, obtained a union of them, and then measured how much of the union grammar was covered by each tool. Figure 3 shows the results. Overall, AsFuzzer achieved the highest coverage, close to 100%, for all the assemblers. We note that both TTT and L* achieve significantly lower coverage than AsFuzzer for Intel and ARM assemblers, whose grammars are more complex than those of other architectures. For instance, the `valignd` instruction in x86-64 requires four operands but the PAC oracle was not able to find a counterexample via random sampling, which resulted in the failure of TTT to infer the valid instruction format for the instruction. This result suggests that the stochastic approach does not work effectively when counterexamples are sparse. There were several cases where AsFuzzer failed to identify valid instruction formats that the other tools could infer. This deficiency originates from a misleading error message from assemblers that incorrectly indicated wrong operand counts. Nevertheless, AsFuzzer achieved significantly higher coverage compared to L* and TTT, which suggests that AsFuzzer is more effective in inferring assembly grammar rules.

4.3 Comparing Test Case Generation Capability

We compared AsFuzzer against Csmith, DifuzzRTL, and Grammarinator to evaluate the test case generation capabilities by generating 1M assembly instructions with each tool and comparing opcode



(a) x86 Clang. (b) x86 GAS. (c) RISCV64 Clang. (d) RISCV64 GAS.

Figure 4: AsFuzzer vs. test case generators.

coverage. We modified AsFuzzer to stop after generating 1M random assembly instructions, skipping its regular fuzzing process. We used Clang and GAS assemblers for x86 and RISCV64 architectures, but we used only RISCV64 assemblers to compare DifuzzRTL as it only supports RISCV64. We denote the combination of a tool and an assembler with a hyphen, e.g., AsFuzzer-Clang means AsFuzzer that uses Clang as an assembler to infer the grammar and generate assembly instructions.

Figure 4 shows the opcode coverage of each tool. AsFuzzer significantly outperforms all the other tools in terms of opcode coverage. Csmith achieved the lowest opcode coverage among them, which clearly indicates that C source-based test case generation is not effective in testing assemblers. This result confirms our motivation that we need a dedicated tool for testing assemblers.

We also note that pre-defined assembly grammars do *not* help much to achieve high opcode coverage. Specifically, the experimental results confirm that the assembly grammars defined in ANTLR [36] are not comprehensive enough to generate diverse assembly instructions. We also observed that over 40.6% of instructions generated from Grammarinator were invalid due to the lack of precision of the grammar rules defined in ANTLR. Figure 4c and Figure 4d also show that manually written grammar rules of DifuzzRTL are not as effective as the automatically inferred grammar rules of AsFuzzer in generating diverse assembly instructions.

4.4 Bug Finding

We now evaluate the effectiveness of AsFuzzer in terms of its bug finding ability. In this experiment, we ran AsFuzzer with a total of 24 different configurations with four different N (= 1, 5, 10, 50) and six different architectures (x86, x86-64, ARMv7, AArch64, MIPS, and RISCV64). To test Intel (x86 and x86-64) assemblers, we ran AsFuzzer with the four target assemblers altogether as they all support Intel architectures. To test the other assemblers of different architectures, we ran AsFuzzer with Clang and GAS only. For each configuration, we ran AsFuzzer for 6 hours.

Table 5: Bugs found by AsFuzzer on different assemblers after a 6-hour fuzzing campaign for each different configuration.

N	Assembler	x86			x86-64			ARMv7			AArch64			MIPS			RISCV64		
		All	Op	Fin	All	Op	Fin	All	Op	Fin	All	Op	Fin	All	Op	Fin	All	Op	Fin
1	Clang	971	168	62	944	134	72	22,238	67	3	1,262	8	0	81,715	126	5	1,324	10	0
	GAS	272	82	21	320	97	27	21,185	69	1	6,132	73	3	77,038	91	4	1,558	11	4
	ICC	532	198	117	323	97	26	-	-	-	-	-	-	-	-	-	-	-	-
	MASM	375	47	36	620	86	48	-	-	-	-	-	-	-	-	-	-	-	-
5	Clang	1,341	223	62	1,214	150	72	34,845	67	3	1,493	8	0	80,881	127	5	2,820	10	0
	GAS	502	97	22	542	111	27	37,646	69	1	12,625	73	3	78,967	98	4	3,558	11	4
	ICC	1,418	255	178	549	111	27	-	-	-	-	-	-	-	-	-	-	-	-
	MASM	442	53	36	799	92	48	-	-	-	-	-	-	-	-	-	-	-	-
10	Clang	1,513	230	62	1,293	152	72	36,456	67	3	1,562	8	0	78,832	126	5	3,419	10	0
	GAS	530	98	21	606	113	27	41,661	69	1	14,061	73	3	77,036	97	4	4,441	11	4
	ICC	2,029	257	182	617	112	27	-	-	-	-	-	-	-	-	-	-	-	-
	MASM	464	54	36	876	93	48	-	-	-	-	-	-	-	-	-	-	-	-
50	Clang	1,940	245	62	1,404	163	72	37,260	67	3	1,531	8	0	76,493	126	5	4,195	10	0
	GAS	624	110	22	648	118	25	43,412	69	1	14,216	73	3	76,286	97	4	5,372	11	4
	ICC	3,085	266	182	691	118	27	-	-	-	-	-	-	-	-	-	-	-	-
	MASM	483	53	36	887	93	48	-	-	-	-	-	-	-	-	-	-	-	-

Op: # of bugs found, grouped by their opcode.
Fin: Final # of bugs found after manual triage.

Table 5 summarizes the number of buggy opcodes found by AsFuzzer for each configuration. The "All" column for each architecture shows the total number of buggy test cases reported by AsFuzzer for each assembler for each configuration. The "Op" column shows the number of buggy test cases grouped by their opcode for each assembler. Finally, the "Fin" column shows the total number of bugs found after manually inspecting their root causes. We reported all the bugs found to the developers.

The total numbers of buggy test cases generated by AsFuzzer are significantly larger than those after manual inspection because of two reasons: (1) some bugs are caused by the same root cause even if they are triggered from different instructions; and (2) AsFuzzer can produce false positives. First, AsFuzzer produces many different assembly instructions of the same format, which can trigger the same bug. For example, the same bug shown in Figure 1 can be found by varying registers and immediate values. Second, AsFuzzer can produce false positives because there are subtle differences between assembly instructions given as input and the machine code obtained from the assemblers. For example, GAS would put dummy instructions when the total size of the output binary is not a multiple of 16 bytes, while Clang would not. This will produce significant differences in the machine code even if given the input assembly instructions are the same. This phenomenon explains the unusually large numbers for RISC architectures, i.e., ARM, MIPS, and RISCV64 assemblers.

To effectively triage the bugs found, we first filtered out most of the buggy test cases by their opcodes (i.e., the "Op" columns). This way we may lose some bugs that are triggered by the same opcode, but we can significantly reduce the number of test cases to inspect manually. We then manually analyzed the root causes of the remaining bugs while examining their source code (if available) as well as the architecture reference manuals to fill out the "Fin" columns. It is noteworthy that one could leverage existing bug isolation techniques [3, 10, 20] to further reduce the manual effort. However, we are not aware of assembler-specific techniques and developing such techniques is beyond the scope of this paper.

From the results, we observe the following two findings: (1) the parameter N does not affect the number of bugs found significantly, and (2) both the consistency check and differential testing mechanisms of AsFuzzer are effective in finding real-world bugs from assemblers.

4.4.1 *Impact of N.*

Recall from §3.1 that N decides how many instructions we generate in each iteration of the fuzzing process for a selected opcode. Thus, setting this parameter too low will generate assembly files with too few instructions, making the overall throughput of AsFuzzer low. On the other hand, setting this parameter too high will allocate too much time to generate assembly instructions for a single opcode, thereby reducing the overall throughput of AsFuzzer. We observed that by increasing N from 1 to 50, the number of buggy test cases generated gradually increases. However, the triaged number of bugs (i.e., the "Fin" column) does not change significantly as we change N because many buggy instructions share the common root cause. Although there is a slight difference in the numbers, we found that it is mainly due to the randomness of the fuzzing process.

4.4.2 *Impact of Two Testing Methods.*

We further investigated the results to see which of the two testing methods (`diff_two_asms()` (§3.3.1) and `diff_asm_disasm()` (§3.3.2) is more effective in finding bugs. Figure 5 compares the numbers of bugs found (after manual triage) by each method. As a result, the consistency check method found 9× more buggy opcodes than the differential testing method. Out of 497 we found, 448 were from the consistency check method.

4.4.3 *Manual Bug Triage.*

We manually triaged the buggy opcodes found by AsFuzzer and grouped them into six major categories as shown in Table 6. (**C1**) There were cases where the assembler silently changed a register into another one as described in §4.5.1. (**C2**) Assemblers often misinterpret a register/immediate/memory operand as a label. (**C3**) Assemblers often ignore pointer directives, thereby producing instructions that have wrong memory access sizes. (**C4**) Assemblers sometimes accept or produce an incorrect number of operands as described in Figure 1. (**C5**) There were

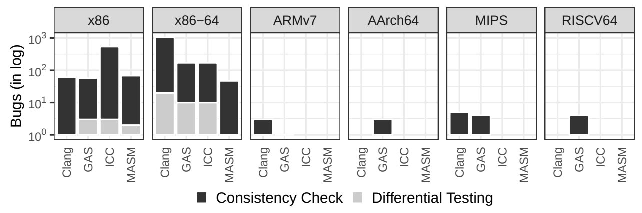


Figure 5: Number of buggy opcodes found for each assembler.

Table 6: Categories of bugs that AsFuzzer found.

	Clang	GAS	ICC	MASM
C1. Using wrong register(s)	23	18	11	11
C2. Confusing an operand and a label	52	19	8	0
C3. Ignoring pointer directives	47	23	22	73
C4. Incorrect # of operands	0	1	169	0
C5. Emitting invalid code	17	0	0	0
C6. Emitting nothing	3	0	0	0
Total	142	61	210	84

several cases from Clang where it generates invalid binary code for valid instructions. (**C6**) There were several cases from Clang where it accepts a wrong instruction and silently produces nothing as shown in §4.5.2. We reported our findings to the developers, and 110 buggy opcodes (**C1**: 20, **C2**: 5, **C4**: 65, **C5**: 17, **C6**: 3) have been confirmed and 23 of them have been fixed by the developers at the time of writing this paper.

4.5 Case Study

Recall that we have already demonstrated a bug found by AsFuzzer with Figure 1 in §2.2. In this section, we present two additional bug cases found by AsFuzzer, which highlight the importance of assembler-specific grammars in testing assemblers.

4.5.1 Case 1: Single Instruction Introducing Four Bugs in Four Assemblers. Figure 6 demonstrates four bugs in four different assemblers found by AsFuzzer with a single assembly instruction. The `lar` instruction in x86-64 can only take a 16- or 32-bit register as the second operand, according to the manual [24]. However, AsFuzzer found that all the assemblers we tested accept a 64-bit register as the second operand, such as `R11`. As a result, AsFuzzer was able to generate the assembly instruction: `lar R11, R12`. Although this is an invalid instruction, all the assemblers we tested accept it and emit a valid (although different than the original) machine instruction as shown on the right side of Figure 6. Note that such an invalid instruction cannot be generated by existing tools because they do *not* consider such an invalid instruction as a test case. It is also noteworthy that we were able to detect four different bugs in four different assemblers with a single test case.

4.5.2 Case 2: Emitting Nothing vs. Rejecting. Figure 7 presents a bug found by AsFuzzer in AArch64 assemblers. The `dsb` instruction in AArch64 takes a special option symbol as its operand, such as `sy`, `st`, etc. However, AsFuzzer found that Clang accepts `dsb` instructions with a regular memory operand, such as "`dsb [R3, #1]`". Although Clang accepts such an instruction, it does not emit any machine code for it. So it will silently ignore the given instruction. On the other hand, GAS rejects such an instruction with an error message. Note that this is a subtle bug that cannot be found by existing tools because it is unlikely that an assembly language model will contain such an invalid instruction.

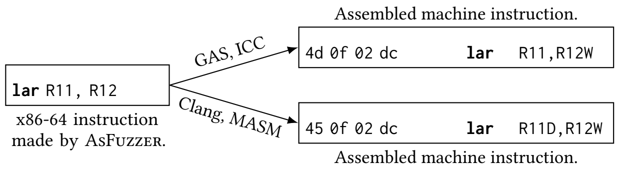

Assembled machine instruction.
GAS, ICC
4d 0f 02 dc lar R11,R12W
lar R11, R12
x86-64 instruction made by AsFuzzer.
Clang, MASM
45 0f 02 dc lar R11D,R12W
Assembled machine instruction.

Figure 6: (Case 1) Previously unseen bugs found by AsFuzzer. All four assemblers accept the wrong instruction.

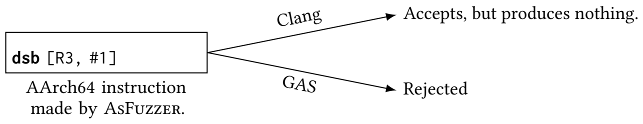

Clang
Accepts, but produces nothing.
dsb [R3, #1]
AArch64 instruction made by AsFuzzer.
GAS
Rejected

Figure 7: (Case 2) Previously unseen bug found by AsFuzzer. Clang accepts the wrong instruction, and then produces nothing, while GAS simply rejects it.

5 Related Work

Compiler Testing. Existing compiler testing approaches can be categorized into two classes: *differential testing* and *metamorphic testing*. Differential testing approaches compare the outputs of two or more compilers (with varying compiler versions and options) [38, 41, 49, 52, 55, 59, 60]. Metamorphic testing approaches generate two semantically equivalent programs and compile them with a single compiler to compare the outputs [13, 37, 56]. Ours follows the differential testing approach but we exclusively focus on testing the assembler of a compiler pipeline, which has not been studied in previous work.

To the best of our knowledge, there is only one prior work that tests assemblers directly [16], which suggests a metamorphic testing approach. However, their approach is specific to IBM's HLASM assembly language, and their main focus is not on assembly instruction generation. Hence, their approach is orthogonal to ours.

Assembly Synthesis. There has been a line of work on synthesizing assembly programs from a given specification. McSynth [53] employs a counter-example-guided interactive synthesis technique to automatically synthesize assembly instructions from a semantic specification. Assuage [22] is an interactive system to help users synthesize assembly programs from a given specification. However, these approaches require a specification of the desired assembly language unlike ours. Moreover, they cannot produce implementation-specific assembly constructs, making them less effective in finding bugs from assemblers.

Emulator and CPU Testing. Since emulators and CPUs take machine instructions as input, testing them is closely related to testing assemblers. EmuFuzzer [46] presents a differential testing approach for finding bugs in CPU emulators. The idea is to compare the states of a physical CPU and a CPU emulator before and after executing the same instructions. Several follow-up works [27, 45] explore similar ideas to find emulator bugs but we cannot directly apply these ideas to our problem since they directly produce machine instructions as outputs but not assembly programs. One exception is KEmuFuzzer [47], which generates assembly programs based on assembly templates but the templates are not publicly available. MeanDiff [32] tests binary lifters with a symbolic differential testing approach, which is closely related to emulator testing as a binary lifter is a core component of an emulator, such as QEMU [8]. However, it also generates machine instructions as a test case, and thus cannot be directly applied to our problem. Similarly, recent CPU fuzzers [9, 23, 28, 29, 34] discover bugs in RTL implementations of CPUs by generating machine instructions. While DifuzzRTL [23] and TheHuzz [29] generate their test cases in the form of assembly programs, they do not consider assembler-specific grammars, making it less effective in finding bugs from assemblers as discussed in §2.2

Grammar Inference. There have been numerous input grammar inference approaches. AUTOGRAM [21] infers a context-free grammar of a program by observing the data flow of each input character at runtime. REINAM [58] symbolically executes the target program to generate seed inputs, and then infers a probabilistic context-free grammar using these inputs. Mimid [15] recovers parse trees by observing dynamic control flows. As these approaches require source code to operate, one cannot use them to infer grammars from commercial assemblers, such as MASM.

On the other hand, there have been black-box approaches [7, 33] that do not require source-level instrumentation. Glade [7] considers a program as an oracle and gradually infers a context-free grammar by observing the behavior of the program given example seed inputs. While this approach is directly applicable to our problem, by replacing our INFERRER module, it suffers from a high computation complexity of $O(n^4)$, where n is the total length of the initial seed inputs. Arvada [33] builds a maximally generalized grammar from the given set of inputs which are accepted by the target program. While it shows performance improvement compared to Glade, the complexity of GETBUBBLES is still $O(n^4)$, which makes directly applying Arvada to assembly grammar inference impractical. On the other hand, our algorithm utilizes useful information from assembler error messages, enabling significantly faster assembly grammar inference.

Unlike the aforementioned grammar inference approaches designed for context-free grammar, there are also regular language inference works [4, 25, 50]. L* [4] and TTT [25] infer regular language with two kinds of oracles: membership and equivalence oracles, but obtaining equivalence oracles in practice is challenging. Instead of equivalence oracles, we can employ stochastic equivalence testing as described by Angluin [5]. However, as discussed in §4.2, we observed that such a stochastic approach suffers in learning assembly grammars, particularly when the valid instruction formats are sparse within the entire set of possible operand combinations. On the other hand, RPNI [50] algorithm utilizes both positive and negative examples to infer regular languages. Nonetheless, it also has a strong assumption of acquiring negative examples [54].

6 Discussion

Recall from §3.2.2, AsFUZZER leverages predefined operand types to reduce its search space by considering only one operand instance per type. However, this approach may miss some syntaxes that require specific operand instances. For example, there are some assembly instructions that only accept a specific register as an operand: when `shl` on `x86-64` takes two operands, the second operand should always be the `cl` register. If we simply consider a random register as the second operand, we may miss this particular syntax, and hence, we may miss some bugs. Extending our approach to consider such cases is a promising future work.

Currently, we construct sets of predefined operand types for each architecture by manually inspecting the architecture reference manuals. Although this is a one-time cost, it is still a tedious and error-prone process. Fully automating this process is indeed a promising future work.

We argue that assembler-specific grammar inference is essential for finding assembler bugs. One may be able to extract a complete grammar from instruction manuals, but it does not reflect the actual grammar implemented by each assembler. As our experimental results in §4.3 show, relying solely on grammars can significantly limit the coverage of assembler fuzzing. Nevertheless, we believe one can leverage available grammars for identifying unrecognized assembly syntaxes that AsFUZZER may miss.

We use a reference disassembler, i.e., `objdump`, in many parts of our system, including the pseudo instruction filtering (§3.2.3), the differential testing (§3.3.1), and the inconsistency checking part (§3.3.2). All these steps assume that our disassembler is correct, but it is not always true. When the disassembler is buggy, our analysis may produce both false positives and false negatives. One may leverage multiple disassemblers to mitigate this problem, but it is beyond the scope of this paper.

7 Conclusion

In this paper, we introduced AsFUZZER, a tool for finding bugs in assemblers. AsFUZZER first infers assembler-specific grammars from the assemblers under test with a novel error-message-driven approach. Our grammar inference algorithm does not require any heavy-cost analysis, and significantly reduces the search space compared to the previous black-box approaches. AsFUZZER successfully inferred grammars from four popular assemblers, including two proprietary assemblers ICC and MASM. With the inferred grammars, AsFUZZER successfully found 497 previously unknown bugs from the four assemblers: 142, 61, 210, and 84 bugs from Clang, GAS, ICC, and MASM, respectively. We reported them to the developers.

Data Availability

Our tool is available at https://github.com/SoftSec-KAIST/AsFuzzer or via Zenodo [30].

Acknowledgements

We thank anonymous reviewers for their invaluable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

References

[1] 2024. ANTLR MASM Grammar. https://github.com/antlr/grammars-v4/tree/master/asm/masm.

[2] 2024. ANTLR RISCV Assembler Grammar. https://github.com/antlr/grammars-v4/tree/master/asm/asmRISCV.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the International Conference on Automated Software Engineering*. 88–99. https://doi.org/10.1109/ASE.2009.25

[4] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[5] Dana Angluin. 1988. Queries and concept learning. *Machine learning* 2 (1988), 319–342. https://doi.org/10.1007/bf00116828

[6] ARM. 2018. ARM® Architecture Reference Manual. https://developer.arm.com/documentation/ddi0406/latest/.

[7] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 95–110. https://doi.org/10.1145/3062341.3062349

[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*. 41–46.

[9] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. HyPFuzz: Formal-Assisted Processor Fuzzing. In *Proceedings of the USENIX Security Symposium*. 1361–1378.

[10] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 223–234. https://doi.org/10.1145/3338906.3338957

[11] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *Proceedings of the International Conference on Software Engineering*. 180–190. https://doi.org/10.1145/2884781.2884878

[12] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (2020), 1–36. https://doi.org/10.1145/3363562

[13] Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic Testing for (Graphics) Compilers. In *Proceedings of the International Workshop on Metamorphic Testing*. 44–47. https://doi.org/10.1145/2896971.2896978

[14] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 206–215.

[15] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 172–183.

[16] Aynel Gül and Vadim Zaytsev. 2019. Mutative Fuzzing for an Assembler Compiler. In *Proceedings of the Belgium-Netherlands Software Evolution Workshop*.

[17] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-Version Compiler Validation. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 191–201. https://doi.org/10.1145/2491411.2491442

[18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a Grammar-based Open Source Fuzzer. In *Proceedings of the 9th SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 45–48. https://doi.org/10.1145/3278186.3278193

[19] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium*. 445–458.

[20] Josie Holmes and Alex Groce. 2018. Causal Distance-Metric-Based Assistance for Debugging after Compiler Fuzzing. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*. 166–177. https://doi.org/10.1109/ISSRE.2018.00027

[21] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the International Conference on Automated Software Engineering*. 720–725.

[22] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *Proceedings of the Annual ACM Symposium on User Interface Software and Technology*. 134–148. https://doi.org/10.1145/3472749.3474740

[23] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1286–1303. https://doi.org/10.1109/SP40001.2021.00103

[24] Intel Corporation. [n. d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.

[25] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In *Runtime Verification, Lecture Notes in Computer Science*, Vol. 96. Elsevier, Amsterdam, Netherlands, 307–322. https://doi.org/10.1007/978-3-319-11164-3_26

[26] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-source LearnLib: A Framework for Active Automata Learning. In *Computer Aided Verification*. Springer, San Francisco, CA, USA, 487–495. https://doi.org/10.1007/978-3-319-21690-4_32

[27] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. Examiner: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 846–858. https://doi.org/10.1145/3503222.3507736

[28] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. 2021. Effective Processor Verification with Logic Fuzzer Enhanced Co-Simulation. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*. 667–678. https://doi.org/10.1145/3466752.3480092

[29] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *Proceedings of the USENIX Security Symposium*. 3219–3236.

[30] Hyungseok Kim, Soomin Kim, Jungwoo Lee, and Sang Kil Cha. 2024. Artifact for "AsFuzzer: Differential Testing of Assemblers with Error-Driven Grammar Inference". https://doi.org/10.5281/zenodo.12786604.

[31] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. 2023. Reassembly is Hard: A Reflection on Challenges and Strategies. In *Proceedings of the USENIX Security Symposium*. 1469–1486.

[32] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungil Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *Proceedings of the International Conference on Automated Software Engineering*. 353–364. https://doi.org/10.1109/ASE.2017.8115648

[33] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning highly recursive input grammars. In *Proceedings of the International Conference on Automated Software Engineering*. 456–467. https://doi.org/10.1109/ase51524.2021.9678879

[34] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. https://doi.org/10.1145/3240765.3240842

[35] Chris Lattner. 2010. Intro to the LLVM MC Project. https://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html.

[36] Chris Lattner. 2024. ANTLR v4. https://github.com/antlr/antlr4.

[37] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. *ACM SIGPLAN Notices* 49, 6 (june 2014), 216–226. https://doi.org/10.1145/2594291.2594334

[38] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the International Symposium on Software Testing and Analysis*. 327–337. https://doi.org/10.1145/2771783.2771785

[39] Xuan-Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. Saffron: Adaptive Grammar-Based Fuzzing for Worst-Case Analysis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019). https://doi.org/10.1145/3364452.3364455

[40] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Fuzzer. In *Proceedings of the USENIX Security Symposium*. 2613–2630.

[41] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 238–251. https://doi.org/10.1145/3582016.3582053

[42] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[43] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 196:1–196:25. https://doi.org/10.1145/3428264

[44] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[45] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-Exploration Lifting: Hi-Fi Tests for Lo-Fi Emulators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 337–348.

[46] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the International Symposium on Software Testing and Analysis*. 261–272. https://doi.org/10.1145/1572272.1572303

[47] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the International*

Symposium on Software Testing and Analysis. 171–182. https://doi.org/10.1145/1831708.1831730

[48] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[49] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 187–196. https://doi.org/10.1145/2491956.2491967

[50] José Oncina and Pedro Garcia. 1993. Identifying regular languages in polynomial time. In *Advances In Structural And Syntactic Pattern Recognition*. 99–108. https://doi.org/10.1142/9789812797919_0007

[51] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the International Symposium on Software Testing and Analysis*. 265–274.

[52] Flash Sheridan. 2007. Practical Testing of a C99 Compiler Using Output Comparison. *Softw. Pract. Exper.* 37, 14 (nov 2007), 1475–1488. https://doi.org/10.1002/spe.812

[53] Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of Machine Code from Semantics. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 596–607. https://doi.org/10.1145/2737924.2737960

[54] Andrew Stevenson and James R. Cordy. 2014. A survey of grammatical inference in software engineering. *Science of Computer Programming* 96 (2014), 444–459. https://doi.org/10.1016/j.scico.2014.05.008

[55] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the International Conference on Software Engineering*. 203–213. https://doi.org/10.1145/2884781.2884879

[56] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *Proceedings of the Asia Pacific Software Engineering Conference*. 270–279. https://doi.org/10.1109/APSEC.2010.39

[57] Leslie G Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (1984), 1134–1142. https://doi.org/10.1145/1968.1972

[58] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. REINAM: Reinforcement Learning for Input-Grammar Inference. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 488–498. https://doi.org/10.1145/3338906.3338958

[59] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 283–294.

[60] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 347–361. https://doi.org/10.1145/3062341.3062379

Received 2024-04-12; accepted 2024-07-03