Generator-Based Fuzzers with Type-Based Targeted Mutation

Soha Hussein
University of Minnesota
Minnesota, USA
soha@umn.edu
Ain Shams University
Cairo, Egypt
soha.hussein@cis.asu.edu.eg

Stephen McCamant
University of Minnesota
Minnesota, USA
mccamant@cs.umn.edu

Mike Whalen
University of Minnesota
Minnesota, USA
mwwhalen@umn.edu

June 5, 2024

Abstract

As with any fuzzer, directing Generator-Based Fuzzers (GBF) to reach particular code targets can increase the fuzzer's effectiveness. In previous work, coverage-guided fuzzers used a mix of static analysis, taint analysis, and constraint-solving approaches to address this problem. However, none of these techniques were particularly crafted for GBF where input generators are used to construct program inputs. The observation is that input generators carry information about the input structure that is naturally present through the typing composition of the program input.

In this paper, we introduce a type-based mutation heuristic, along with constant string lookup, for Java GBF. Our key intuition is that if one can identify which sub-part (types) of the input will likely influence the branching decision, then focusing on mutating the choices of the generators constructing these types is likely to achieve the desired coverages. We used our technique to fuzz AWSLambda applications. Results compared to a baseline GBF tool show an almost 20% average improvement in application coverage, and larger improvements when third-party code is included.

1 Introduction

Fuzzing (i.e., large-scale automated random testing) can be one of the most practical ways to find bugs that might otherwise be missed by developers.

Coverage-guided fuzzing (hereafter "CGF"), as popularized in the tool AFL [1], augments mutational fuzzing by recording which code locations a test input covers. CGF then prioritizes inputs that cover previously-uncovered code (intuitively, "interesting inputs") as the starting points for future mutations. CGF is effective at biasing the search process towards relevant inputs without requiring expensive analysis, and this power has made it popular both for practical testing and as the basis for research on further enhancements [2, 3, 4, 5, 6, 7, 8].

Nevertheless, CGF's effectiveness is impaired when used to generate structured inputs. This is because traditional CGF lacks information about the input's structure, leading to the generation of largely syntactically incorrect tests that typically fail to reach the deeper parts of the program's code. One potential solution to this problem is to use input generators. An input generator for a given type (also commonly used in property-based testing such as QuickCheck [9]) is simply a function that returns a random instance of the type. Using generators to provide inputs yields generator-based fuzzing (hereafter "GBF"). GBF can be used for both unit and system testing, and can capture patterns that constitute legal and typical inputs. The effort of writing generators can be amortized because many data types (e.g., associated with common file formats or libraries) are reused across applications, and generators for composite types can be expressed in terms of generators for their components. More recently, the authors of the Zest system [10] demonstrated how CGF and GBF approaches can be combined by applying mutations to the sequence of random choices a generator uses to build objects used as program input. In more detail, generator-based fuzzers work by recording and mutating the sequence of choices made by random-number interfaces within the generator code. We call this sequence of choices the *fuzzer chosen input* or FCI, as distinct from the outputs of the generator methods which are the inputs to the program under test. Such mutations are more likely to achieve semantically relevant coverage because the generators enforce input validity constraints. However even when CGF and GBF are combined as in Zest, the location for a mutation is still chosen uniformly at random across the input. We observe a missed opportunity that such tools still have no connection between their coverage goals and the process of selecting mutations.

Moreover, more precise targeting of mutations becomes increasingly important as test inputs become larger. On a small input, choosing mutation locations uniformly at random does not require much time to cover every part of an input, just by brute force. But for larger inputs, the most valuable locations to mutate shrink as a fraction of the input size, such as if a format's metadata has a fixed size while the data it describes gets larger. For large inputs, selecting mutation locations uniformly at random can be wasteful.

While more expensive analysis techniques such as taint tracking or symbolic execution could also be used to trace which parts of the input are relevant to code [11, 12, 13], but they are unlikely to scale. This is because tracking the complete data flow in a cross-language and distributed environment as found in modern applications, i.e., serverless computing, would be very difficult.

Similarly, being selective about relevant code becomes more important when fuzzing larger programs that make extensive use of libraries and frameworks. Libraries and frameworks improve productivity what they encapsulate rich functionality and allow the code that uses them to be relatively simple and high-level. But in this case, the code newly written for an application may be only a small fraction of the total code executed. We will use the term "application" to refer generically to the often smaller code that was written for a specific use case and that is the primary target of fuzzing. We will use "third-party code" to refer generically to all of the other code that also executes as part of the program under test but is not the primary target of fuzzing, such as libraries and frameworks. If a fuzzing tool does not differentiate third-party code, it can overwhelm application code, for instance if third-party code contributes most of the code coverage obligations in CGF.

To scale fuzzing as the input size and third-party code usage become significant, fuzzers must concentrate their effort to exercising branches in the application by targeting the most relevant parts of the input.

In this paper, we present a novel technique that can focus the fuzzer to cover application code, by mutating substructures in the input that is likely to trigger the intended code target. The core idea of our approach is to link input generation to code in the application that uses that input via types.

This approach is well-suited for a statically-typed object oriented language like Java, where type information is directly available in the bytecode (or source code) for both input generators and for branches in the program under test. We implement a static analysis that, for each branch in the program under test, determines a ranked list of *influencing types* of objects that are data-flow predecessors of the branch condition.

As the other half of the connection, our approach associates types with segments of the input, namely the return types of input generator methods. Since generator methods commonly call other generator methods, our implementation uses a simple form of execution indexing [14] to record the type information for input-generating code in the context of its call stack. Our system uses execution indexing to associate a set of generated types with a segment of the FCI, in a way that maintains that association even if the execution of other generation methods changes.

Our system then uses the matching between types statically associated with branches, and types dynamically associated with parts of the input, to increase the frequency of mutating parts of the input corresponding to objects that influence uncovered branch obligations. As an additional improvement to accelerate the fuzzing process, our static analysis also collects constant strings from the application code, and includes these strings as possibilities when generating strings in program inputs.

For our main evaluation, we fuzz software for serverless microservices written in Java for AWS Lambda and compatible platforms. This is a practically important area where it is common for a small piece of application software to depend on a complex stack of third-party libraries and frameworks. The serverless domain also illustrates why it is appealing to use a lightweight fuzzing technique that does not require entire data flows and software stacks to be instrumented.

The results show that our technique improves the code coverage achieved by fuzzing. Coverage of application branches improved by an average of 18.2% (geometric mean) over the baseline, and coverage including third-party code increased by a geometric mean of 43.2% when the set of covered libraries remained the same, and even more when our approach covers additional libraries. We also separately measure runtime and memory overhead.

The paper is organized as follows: Section 2 provides an example, Section 3 outlines our technique, Section 4 presents our evaluation, Section 5 discusses related work, and Section 6 concludes.

```java
1.import java.util.List;
2.import javax.imageio.ImageIO;
3.import java.util.GregorianCalendar;
4.import java.nio.file.attribute.BasicFileAttributes;
5.public class YearThumbnails {
6. public static void main(String[] args) {
7.  String inputPath = args[0];
8.  List<File> filteredImages = new ArrayList<>();
9.  File[] imageFiles = new File(inputPath).listFiles();
10.  if (imageFiles != null)
11.   for (File file : imageFiles)
12.    if (yearTaken(file.toPath())== 2008)
13.     filteredImages.add(file);
14.   if (!filteredImages.isEmpty())
15.    regroupInSingleImage(filteredImages); }
```

(a) Program for creating thumbnails for images modified in 2008.

```java
1.private static int yearTaken(Path imagePath) ... {
2. BasicFileAttributes attr = Files
3.   .readAttributes(imagePath, BasicFileAttributes.class);
4. long modifiedTime = attr.lastModTime().toMillis();
5. GregorianCalendar calendar = new GregorianCalendar();
6. calendar.setTimeInMillis(modifiedTime);
7. return calendar.get(GregorianCalendar.YEAR);}
```

(b) Helper method for computing the getting modification year.

Figure 1: A snippet of a program that creates thumbnails of images for a specific year, written with the help of ChatGPT.

2 Motivating Example

Consider the program in Fig. 1a, generated with assistance from ChatGPT [15]. The program creates a mosaic image by combining thumbnails of all images modified in the `year` 2008. Given the location of image files in `inputPath` **(line 7)**, the program first finds all files within the specified path **(line 11)**, then it iterates over all image files within this directory to collect those whose modification date matches the intended year **(lines 11-13)**. The helper function `yearTaken` (in Fig. 1b) finds the system modification year of a file object. The method first attempts to read attributes of the file system for the image path, particularly the last modified time (**line 2-4**. Then, it converts that time into a date form captured in a `GregorianCalendar` object **(line 5-6)**. Finally, the method extracts and returns the year of the last modification of the file **(line 7)**. The thumbnail program in Fig. 1a creates a combined thumbnail image for the filtered images **(line 15)**, using some image processing happening within the `regroupInSingleImage` method, whose details are omitted.

Existing fuzzing tools cannot test this program effectively. This is because this program's input is not explicitly passed to the program, moreover, the program's input is substantially larger than inputs to programs typically tested by traditional fuzzers. For example, in this program, multiple images are needed to test the resizing functionality, whereas traditionally fuzzers are used to test a single image processor program. This relatively large input size of the program makes the random mutation strategy unlikely to construct a useful input, i.e., an image with the year 2008, to achieve the desired coverage. In this case, the input bytes representing the date information are likely much smaller than the actual image. For example, to generate an image file of a 100x100 size, where each pixel needs red, blue, and yellow color values, the size of the FCI will be (100x100x4) values. On the other hand, the modification date of the file is about 26 to 30 values within the FCI. This implies the random mutations attempted by GBF are likely to fall outside of the interesting bytes; i.e., mutations are introduced to the actual content of the image, not its file system's date. Furthermore, observe that the program utilizes some third-party libraries, such as using the `ImageIO` library for image processing. Also, observe that while all code coverage is potentially relevant and useful for fuzzing, prioritizing coverage of the program under test, in this case the `Thumbnail` program without the underlying libraries, can potentially be more practical, leading to faster coverage. Thus GBF needs a mechanism to emphasize coverage of coverage targets within the application code while de-emphasizing the significance of third-party library coverage.

```
1.public class FileWithAttributesGenerator extends ... {
1.@Override
2. public File generate(SourceOfRandomness r, ..) {
3.  String fileName = gen().make(StrGenerator.class).generate(r, ..);
4.  FileGenerator fileGenerator = gen().make(FileGenerator.class);
5.  fileGenerator.setFileName(fileName);
6.  File outFile = fileGenerator.generate(r, ..);
7.  boolean attrChanged = false;
8.  while (!attrChanged) { //repeat until successful change
9.  GregorianCalendar cCal = gen()
10.                  .make(CalGenerator.class).generate(r, ..);
11.  GregorianCalendar aCal = gen()
12.                  .make(CalGenerator.class).generate(r, ..);
13.  GregorianCalendar mCal = gen()
14.                  .make(CalGenerator.class).generate(r, ..);
15. //  code for setting file attributes removed
16.  } //  code for changing file attribute removed
17. return outFile; }
```

(a) Snippet of the file input generator. Invocations of enclosed generators are highlighted in grey.

```
19.public class CalGenerator extends ... {
20. @Override
21. public GregorianCalendar generate(SourceOfRandomness r, .. ) {
22.  GregorianCalendar cal = new GregorianCalendar();
23.  cal.set(DAY_OF_MONTH, r.nextInt(31) + 1);
24.  cal.set(MONTH, r.nextInt(12) + 1);
25.  Integer yearsAway = geoDist.sampleWithMean(5, r) - 1;
26.  if (r.nextFloat() > 0.25)
27.   cal.set(YEAR, Year.now().getValue() + yearsAway);
28.  else
29.   cal.set(YEAR, Year.now().getValue() - yearsAway);
30. //  code for generating a random time and a time-zone removed
31.  return cal; } }
```

(b) Snippet of the calendar generator. Random parametric choices controlled by GBF are highlighted in grey.

Figure 2: A snippet of generators

2.1 GBF Background

One way to fuzz the program in Fig. 1a is by using a generator-based fuzzer, also known as semantic fuzzers [10, 13, 16]. These fuzzers are able to construct structurally correct input using programmable code that we call *input generators*. For example, Fig. 2 shows a snippet of the code for constructing a file generator Fig. 2a `FileWithAttributesGenerator` which uses, among other generators, the calendar date generator 2b, i.e.; `CalendarGenerator`. Both classes define a `generate` method that constructs the object of the intended type. For example, in the `FileWithAttributeGenerator` the `generate` method starts by creating a random file name using the `StrGenerator` **(line 4)**, then it creates an instance of the `FileGenerator`, passes the randomly selected name to it, then generates the file's contents **(lines 5-7)**. Then, the remaining code generates random dates using the `CalendarGenerator` to set the creation date, last access date, and the last modification dates of the created file. The `GregorianCalendar` generator creates a calendar object **(line 22)**. It randomly sets its day and month to randomly generated values **(lines 23, 24)**. Finally, it assigns a year to the calendar using a random number that represents a number of years relative to the current year. The selection process employs a geometric distribution with a probability of 1/5. Subsequently, there is a 0.25 probability that the chosen value is added to the current year, resulting in a choosing the current year or a future one, while in other cases, it's subtracted from the current year, yielding a year in the past.

2.2 Technique Overview

There are two observations on using generators to construct program input. First, observe that the input-generators can naturally be composed together as in the composition of the `StrGenerator` and `CalGenerator` within the `FileWithAttributesGenerator`. Although it is possible to rewrite generators without following this hierarchical arrangement, dividing the object's generation functionality into smaller, more modular components conforms to sound programming principles and is likely to promote better code organization, reusability, and maintainability. Second, observe that GBF does not directly control the program input. Instead, it controls the random selection within the input-generators represented with operations on `r`, the `SourceOfRandomness`. By controlling these random selections, the GBF can indirectly control the generation of program inputs. For example, the GBF fuzzer will control the random selection happening on **line 5-7** in Fig. 2b. GBF can choose between reusing old values or generating fresh random ones, thereby modifying an old input, or constructing a new one.

Making targeted mutations necessitates understanding which parts of the FCI need modification. Directing GBF fuzzers is especially challenging because of the lack of mapping between the FCI and the program's input. Although heavyweight techniques such as taint analysis, symbolic execution, or certain dataflow analyses can be employed to trace the propagation of the FCI (Fuzzer Chosen Input) to the program's input, this process is often complex and resource-intensive, potentially impacting the fuzzer's performance. Additionally, there are scenarios where conducting such analyses may not be feasible due to the presence of heterogeneous languages and systems. For example, it is not easy to track FCI that is going out to a file system.

In this paper, we develop a new type-based mutation heuristic that can direct the mutation to change likely useful subsequences within the FCI based on object types. For example, to cover both arms of the branch at **line 12** in Fig. 1a, one can recognize with a simple static analysis that the object of type `GegorianCalendar` is the most useful part of the input that needs to be mutated to generate a file whose modification year is 2008. Identifying the types impacting a branch is computed with our *influencing type analysis*. Then, during the construction of the program input, every byte within the FCI is mapped to the type of objects they are used in generating. For example, suppose $v_1 = 16$ and $v_2 = 8$ are parts of FCI used **(lines 23, 24)** in Fig. 2b. Then, one can identify that the dynamic types that v_1 and v_2 are part of the `GregorianCalendar` and `File` types. This can be computed by looking at the generator's callstack at the point of generating a random number for these expressions. We call these types the *dynamic types*. During the fuzzer's operation, our extension maintains a mapping between the bytes of the FCI and their dynamic types.

During the mutation process, our extension correlates the types influencing the uncovered code targets with the dynamic types linked to the FCI. Subsequently, employing selection heuristics outlined in the subsequent section 3, if a type match is identified, our extension prioritizes mutating the subsequences within the FCI that correspond to the matched type.

We ran the baseline fuzzer and our extension over the program in Fig. 1a. Each run is an hour long and we repeated the experiment 20 times. Our results show that the baseline fuzzing tool was able to cover **line 13** 35% of the time (7 out of 20), while our fuzzing extension achieved the same coverage with a success rate of 70% (14 out of 20), with a median 1201.05s (mean 217.9s) versus 998.71s (mean 377s) for the baseline fuzzing tool and our fuzzing extension, showing a skewed probability distribution.

3 Technique

Alg. 1 shows the main algorithm of GBF, with our extension highlighted in grey. The algorithm takes two inputs: an input generator g and a program p. The fuzzer provides the fuzzer-controlled input from the set of FCI I_c to the generator g which returns a program input from the set of inputs I_p. Program p is tested on input from I_p, then terminates with either a FAILURE or a SUCCESS result[1]. With V, and $\mathcal{T}$ representing the set of random byte values and the set of all types, respectively, the GBF defines the following four data structures: (1) the set of fuzzer-controlled input I_c, which is defined as a sequence of random values, (2-3) the set of successful and failed input $\mathcal{S}, \mathcal{E}$, and (4) The set of all code coverage $\mathcal{C}_{total}$.

GBF proceeds as follows. Initially, the set of inputs is initialized with a single randomly-chosen FCI **(line 15)**. Also, the failed inputs and the set of total coverages are set to the empty sets **(line 15)**. Then, the algorithm proceeds in the fuzzing process until a specified time budget expires (**lines 16-30**). At each step, one input i_c is picked up from $\mathcal{S}$, then the number of candidates to be generated from i_c is determined using the NUMCANDIDATES heuristic **(line 18)**. In each iteration, the selected input is mutated to generate i'_c **(line 19)**. Mutation means introducing random changes to random locations within i_c. Next, the generator is invoked on the mutated inputs **(line 21)**, then GBF runs the program p on the constructed program input i_p, and collects the result and the coverage information **(line 22)**. If the program fails, the program input i_c is added to the set of failed inputs **(lines 23-24)**. Otherwise, if new coverage was achieved, GBF saves i_c in the successful inputs and updates its total coverage **(lines 26-29)**.

Our type-based targeted-mutation extension modifies the definition of the fuzzer-controlled input I_c **(line 5)**. That is, in our extension the fuzzer-controlled is defined by the product type of the randomly generated values V, execution indexes EI defining the unique dynamic execution location within the input generators at the point of generating a specified random value, and sequence of types T^* capturing the return value of each method within the callstack at the

[1]We assume that p is always terminating. In the implementation a timeout is imposed for each test

input : A generator g: $I_c \rightarrow I_p$
input : A program $p : I_p \rightarrow \{\text{FAILURE}, \text{SUCCESS}\}$
1 **Type:** Set of random values V
2 **Type:** Set of types $\mathcal{T}$
3 **Type:** Set of execution indexes EI
4 **Type:** Set of fuzzer-controlled inputs $I_c = V^*$
5 **Type:** Set of fuzzer-controlled inputs $I_c = (V, EI, \mathcal{T}^*)^*$
6 **Data:** Set of successful inputs $\mathcal{S} \subseteq I_c$
7 **Data:** Set of failed inputs $\mathcal{E} \subseteq I_c$
8 **Data:** Set of total coverages $\mathcal{C}_{total}$
9 **Data:** Set of all code targets $\mathcal{K}$
10 **Data:** Set of uncovered code targets $\overline{\mathcal{C}}_{\mathcal{T}} \subseteq \mathcal{K}$
11 **Data:** Map of typing distance $\mathcal{D}$: $\mathcal{T} \rightarrow \mathbb{N}$
12 **Data:** Map of target typing distance $\Gamma : \mathcal{K} \rightarrow \mathcal{D}^*$
13 $\Gamma \leftarrow \text{ANALYZE}(p)$
14 $\overline{\mathcal{C}}_{\mathcal{K}} \leftarrow \text{KEYS}(\Gamma)$ $\quad \mathcal{D} \leftarrow \text{UNIFYTYPES}(\Gamma)$
15 $\mathcal{S} \leftarrow \{$ RANDOM $\}$ $\quad \mathcal{E} \leftarrow \emptyset \quad \mathcal{C}_{total} \leftarrow \emptyset$
16 **repeat**
17   **for** i_c *in* $\mathcal{S}$ **do**
18   **for** $1 \leq j \leq \text{NUMCANDIDATES}(i_c)$ **do**
19             $i'_c \leftarrow \text{MUTATE}\ (i_c)$
20             $i'_c, \mathcal{D}' \leftarrow \text{MUTATE}\ (i_c, \overline{\mathcal{C}}_{\mathcal{K}}, \mathcal{D})$
21             $i_p \leftarrow g(i'_c)$
22   coverage, result $\leftarrow \text{RUN}(p, i_p)$
23   **if** *result*=FAILURE **then**
24                 $\mathcal{E} \rightarrow \mathcal{E} \cup \{i_c\}$
25   **else**
26   **if** coverage $\nsubseteq \mathcal{C}$ **then**
27                     $\mathcal{S} \leftarrow \mathcal{S} \cup \{i_c\}$
28                     $\mathcal{C}_{total} \leftarrow \mathcal{C}_{total} \cup$ coverage
29                     $\overline{\mathcal{C}}_{\mathcal{K}} \leftarrow \overline{\mathcal{C}}_{\mathcal{K}} -$ coverage
30 **until** *given time budget expires*
31 **return** $g(\mathcal{S}), g(\mathcal{F})$

Algorithm 1: Generator-based fuzzer algorithm, modelled after Zest [10], and amended with our extension in grey

point of generating the random value. In addition to that, our extension introduces four additional data structures: (1) A set of all application code targets ($\mathcal{K}$), (2) the set of uncovered code targets ($\overline{\mathcal{C}}_{\mathcal{K}}$), (3) the map of typing distance ($\mathcal{D}$), and (4) the map of code targets with their typing distances (Γ).

Our algorithm starts by analyzing the program p. Our analysis collects code targets to populate the influencing types and together with their distances in Γ (**line 13**). Using this map, our extended algorithm initializes the set of uncovered code targets and unifies their typing distances (**line 14**). The distance of a type indicates how far a type is from the code target. The closer the distance of a type the more likely it will be selected by the fuzzer for mutation. On the other hand, the unification of types aims to associate a single distance to each type. This is because the same type can have different distances for different code targets. Note that, mutation does not control which branch it will flip, so when choosing what type of object to mutate, we should choose based on all the branches that a type might be relevant to. This is why we unify type distances among all code targets. Our extension uses a different mutation operation by running the mutation operation over the updated fuzzer-controlled input i_c, the uncovered code targets $\overline{\mathcal{C}}_{\mathcal{K}}$ and the map of typing distances $\mathcal{D}$. The result of this operation is a mutated input (i'_c), and an updated typing distance map $\mathcal{D}'$ (**line 20**). Then, GBF proceeds with running the program on the newly constructed program input, then if new coverage is achieved then the new coverage is removed from the set uncovered code target if matched (**line 29**). We discuss each component in detail in the following sections (Sec.-3.1 - Sec. 3.2).

3.1 Static Phase

In this step, our extended algorithm analyzes the *application* code to find all code targets. The idea is to direct the fuzzing toward covering these code targets, in effect giving more precedence to application code coverage. In our analysis, we define *code target* to be either of the two sides of a branching condition within the application code. Then,

```
1. v17 = arrayload v10[v26]
2. v19 = invokevirtual File.toPath()Path v17
3. v21 = invokestatic YearThumbnails.yearTaken(Path)I v19
4. if(ne,v21,v3)
```

Figure 3: Simplified Wala IR for lines 12 in Fig. 1a. `v10,v17,v3,` and `v26` represent `imageFiles, file`, the value `2008` and the length of `imageFiles`

```
(Ljava/util/GregorianCalendar, 2)
(Ljava/nio/file/attribute/FileTime, 4)
(Ljava/nio/file/attribute/BasicFileAttributes, 5)
(Ljava/nio/file/Path, 6)
(Ljava/io/File, 8)
([Ljava/io/File, 9)
```

Figure 4: A simplified snippet of the influencing types found for the branch at **line 13** in Fig. 1a. Each line shows the computed influencing type and its distance from the decision.

for each code target, we compute types likely to influence their decision. An *influencing type* is any type that can flow to the operands of the code target.

In the general sense, our analysis computes a variation of def-use relation among variables. The analysis processes an IR derived from the bytecode. We use the Wala [17] static analysis framework to collect influencing types for this analysis, whose IR is in a static single-assignment form [18]. For example, Fig. 3 shows a simplified Wala's IR representation of **line 12** in Fig. 1a, where the `imageFiles` is first loaded. This corresponds to loading elements of the array within the `for` loop at **line 11**. Then, at **line 2** the image file is retrieved using `toPath()`, followed by invoking the application method `yearTaken()` to find the year of the photo (**line 3**), then finally branching on the resulting output (**line 4**). The analysis consists of three passes: a *variable dependency pass*, a *type collection pass*, and a *type unification pass*.

- Variable dependency pass: In this pass, we create a dependency graph between variables using a variation of the traditional def-use dataflow. Our extension gathers variable dependencies through a depth-first traversal of the application's call graph. To capture the variable dependency when a method invocation is encountered, we add more edges to the dependency graph to capture the dependencies between the actual and formal parameters of the method. Similarly, if there is a return statement within the method, then we create a dependency between the returned variable in the callee and that of the caller. Also, for non-static methods, we capture the def-use dependency between the variables passed to the arguments and the reference variable on which the method invocation is done. Our analysis can identify recursive calls but visits the method only once at each encounter. Note that, for any invocation of a non-application method, i.e., a library call or a Java standard library call, we avoid analyzing their body. For example, the edges (use $\rightarrow$ def) in the dependency graph for this pass for Fig. 3 are:

$$\{\texttt{v17} \rightarrow \texttt{v10}, \texttt{v17} \rightarrow \texttt{v26}, \texttt{v19} \rightarrow \texttt{v17}, \texttt{v21} \rightarrow \texttt{v19},$$

$$\texttt{farg}(\texttt{yearTaken}) \rightarrow \texttt{v19}, \texttt{v21} \rightarrow \texttt{ret}(\texttt{yearTaken})\} \cup \texttt{E}_{\texttt{yearTaken}}$$

Observe that `v21` also depends on $\texttt{E}_{yearTaken}$: the set of variable-dependency edges computed from analyzing the `yearTaken()` method. By contrast, the `toPath()` analysis is skipped since it is not identified to be defined within the application code (within the user's outermost package name).

Also, in this pass, we find and collect the type for each variable encountered. To find types, we use the static type signature existing in the code as well as Wala's type inference. If a variable was found to have multiple types, we keep all of them. In general, we exclude primitive types and `java.lang` object types.

- Type Collection Pass: In this pass, our extension walks all conditions and computes their influencing types. This is done by walking the variable dependency graph from the previous step for each operand within a branch condition. That is, we traverse the dependency graph backward, collecting encountered variable types and their respective distances. The distance to an influencing type indicates its proximity to the target branch, reflecting its significance to the mutation process. The intuition here is that closer influencing types probably have a larger effect on the branch's decision. Moreover, the closer the influencing type is, the more likely it is to be fine-grained, which means, that the mutation can now happen over a smaller subsequence within the FCI. For example, Fig. 4 shows a simplified subset snippet of the influencing types collected for the branch at **line 12** in Fig. 1a. Each line is a pair of a type and a distance. Note that there might be multiple types at the same distance. In this case, we can see that the influencing type `GregorianCalendar` is much closer than the `File` type. This suggests that changing subsequences within the FCI related to the generation of the `GregorianCalendar` is more likely to make necessary changes to cover both arms of the intended branch. Finally, note that we ignore Java String types, primitive types, and their corresponding wrapper classes, such as Double and Float. These types are too common and do not provide sufficient distinction for targeting specific conditions.

- Type Unification Pass: In this pass, we merge all influencing types with their distances for all code targets into a unified structure ($\mathcal{D}$). If influencing types are unique among all code targets then the unified structure will be the set of all types with their distance. However, if an influencing type appears in multiple code targets with different distances, then the type is added to $\mathcal{D}$ with its distance in the largest code target; i.e., the code target with the largest number of influencing types. The intuition is that the more influencing types the code target has, the more information and a finer distinction about the types exist in this set. For example, if (t, d_1) and (t, d_2) are found in two code targets k_1, and k_2, then the unified structure will contain t, d_3 such that $d_3 = d_1,$ if $|k_1| > |k_2|$, otherwise $d_3 = d_2$, where $|k|$ denotes to the number of influencing types associated with a code target. Later, distances of various influencing types are updated during the dynamic fuzzing process depending on whether a static influencing type could be matched with a dynamic type. The change in distances allows the fuzzer to prioritize or de-prioritize influencing types depending on whether they were useful dynamically.

- Creating the Constant String Lookup Table: We observed that using constant strings from the application code as string values when generating strings in the program input, can improve the fuzzing process. In the implementation, we employ static analysis to gather reachable string constants from the call graph's entry point. Subsequently, during the fuzzing process, we enable the fuzzer to alternate between these constants and generating fresh strings when generating string values.

3.2 Fuzzing Phase

In this step, we use information about influencing types collected from the static phase to make type-targeted mutations. Recall that there is no direct connection between the FCI and the program input. To enable type-directed mutation, one must first establish this connection between the bytes within the FCI and the types of objects they are used in generating. However, annotating the FCI with types is not enough to ensure that bytes after mutation will be used to generate the same object type. This is because mutations in the FCI are likely to lead to a different control flow within the generators. Thus if the FCI is consumed by the input generators only by the order the bytes they appear in, then there is no guarantee that the same byte within the mutated FCI will generate the same object type. One can only achieve that if we keep track of the dynamic locations where bytes within FCI are used. We do this by using *execution indexing*. Execution indexing (EI) [19, 20] of a program provides an ordered, unique representation of the dynamic points within the execution (as with an execution trace). Execution indexing technique has many applications such as program alignment [21], and detecting deadlocks and concurrency failures [22, 23]. The main idea of execution indexing is that every program point is associated with a vector of code points that identify the unique dynamic index of a particular execution point. We use EI to ensure that bytes can only be reused in the same dynamic locations. Next, we outline the process of annotating bytes within the FCI with types, creating their execution indices, and utilizing this information for type-targeted mutations.

3.2.1 Using the Execution Index to annotate the FCI with Types

To annotate the FCI with types, we intercept invocations of requesting a random number, as in `r.nextInt()`. Then, we collect the type of objects that are currently being constructed. We do that by collecting the return type of each method invocation in the current callstack. For example, we intercept the execution of `r.nextInt(31)` in **line 23** in Fig. 2b. Let's assume that the subsequence of the FCI for this random selection is the number **9**. Then, we associate this value with the return types of all method invocations types in the current callstack. In this case, the value 9 will be annotated with {`GregorianCalendar`, `File`} types; i.e., (`9`,{`GregorianCalendar, File`}).

As mentioned before, annotating bytes of the FCI with types is not sufficient for the targeted mutation. This is because without associating each byte within the FCI to their corresponding execution points, after mutation, the same byte can be reused in a different dynamic location, generating a different object type. For example, consider the case where initially generating a `fileName` required that the `StrGenerator` use the first 10 bytes of the FCI. Suppose a mutation occurred that resulted in having `StrGenerator` using only the first 5 bytes of the FCI. In that case, the remaining 5 bytes from the original FCI, while being annotated with `String` types, will be used within the `FileGenerator` to create an object of `File` type. This is a mismatch between the object types created by the same FCI's subsequence from one test to another.

To avoid reusing bytes in the same position in different FCIs, we associate an execution index with every byte of the FCI. We use an execution indexing implementation in Zest [24]. The main idea of the execution index is that every program point is associated with a vector of code points that identify the dynamic EI of a particular execution point. In our extension, we use a relaxed definition of execution indexing [14] implemented in Zest, where the uniqueness

Table 1: List of AWS lambda applications obtained from GitHub with the number of enclosed lines of code (loc)

Serial	**benchmark**	**loc**
1	s3-java	174
2	anishsana	101
3	load-historic-data	182
4	lambda-unzip	170
5	csv-loader	407
6	nikoshen	131
7	upload-survey	16,661

property among different execution indices is not guaranteed. More precisely, let $m_1, m_2, .., m_n$ be the call stack when the execution of an expression p is about to happen. The execution index of p (written as $ei(p)$) is defined as a vector of pairs $[l_i, q_i]$ such that $(0 < i \leq n)$, where l_i is a label of the expression within method m_i, and where q_i is the number of times the statement was invoked during the current invocation of m_i.

To illustrate, consider the input generator at Fig. 2a where a file is created and its creation date is constructed by invoking the `CalGenerator` at **line 13** (Fig. 2b). If `lCal` is created during the first iteration of the while loop on **line 9**, then the execution index of the random choice of selecting a day of a month (**line 23** in Fig. 2b); i.e., $ei(23) = [13, 1, 23, 1]$. However, if `lCal` is created during the second iteration of the while loop then the execution index of the random choice of the day of a month would be $ei(23) = [13, 2, 23, 1]$. This is because the `generate(r,..)` method at **line 13** in this case has been invoked twice in the context of `FileWithAttributesGenerator`.

Using this approach the FCI (I_c) would be represented as a sequence of triples of the form $I_c = (V, EI, \mathcal{T}^*)^*$. For example, continuing our above example of creating the `lCal` object **line 13**. Our extended FCI for selecting the day of the month (**line 23**) will be the triple of the form: `(9, ei(23),{GregorianCalendar, File})`

3.2.2 Type-Based Mutation

In this step, we use the dynamically captured type distances maintained in $\mathcal{K}$ to select a type to mutate. The probability of choosing a type is inversely proportional to its distance. When a type is selected we then match it up with EI objects within the FCI. If an EI object is found with a matched type from the previous step, SpotOn proceeds in two steps. First, SpotOn decreases the type's distance (multiplying it by 3/4). Second, SpotOn randomly mutates a subsequence of a random length starting from the matched EI object. If multiple EI objects were matched, then we randomly chose EI objects to mutate their values. If no EI object was found with a matching type, then the type's distance is updated (multiplying it by 4/3) to decrease the type's probability of being selected in future iterations.

4 Evaluation

Our research questions are:

RQ1: Does the SpotOn achieve more coverage than GBF?

RQ2: What is the effectiveness of the type-based targeting without the constant string optimization?

RQ3: What is the overhead of the type-based targeted mutation?

We evaluated our technique using AWS Lambda applications. These applications tend to be compact in the size of user-written code while using extensive code from third-party libraries. Additionally, their inputs often possess a degree of complexity and structure, albeit loosely defined. We collected our benchmark suite from GitHub. We searched for AWS Java Lambdas that are triggered by an `S3EventNotification` and that use a single `S3`, and/or a single `DynamoDB` service. We removed applications with no branches and those that use unsupported formats.

Tab. 1 lists of benchmarks we used, which include:

1) `s3-java` [25]: when an image file is inserted/updated, the lambda scales it and puts it back in the bucket.

2) `anishsana` [26]: when a CSV file of students' grades is inserted/updated into an `S3` bucket, the lambda computes an average grade and places a record into a DynamoDB table.

3) `load-historic-data` [27]: when a stock ticker CSV file, the lambda collects some data from the file and creates a corresponding record onto a DynamoDB table.

4) `lambda-unzip` [28]: an archive is inserted/updated into an `S3` bucket, the lambda unzips its content into the bucket and removes the archived file.

Table 2: Average application coverage, (+SpotOn): only found only by SpotOn (+Zest): only found by Zest. (*) indicates statically significant results (p-value = 0.05)

benchmark	common	+SpotOn	+Zest
lambda-unzip	21.2	1.80	0
csv-loader*	13.45	4.45	0.7
anishsana	8.45	0.50	0.05
load-historic-data*	13.15	0.80	0.05
nikoshen*	12.60	2.9	0
s3-java	16.65	0.35	0
upload-survey*	10.00	6.10	0

Table 3: Average of *all* (third-pary-included) coverage, (+SpotOn): only found by SpotOn (+Zest): only found by Zest. (*) indicates statically significant results (p-value = 0.05)

benchmark	common	+SpotOn	+Zest
lambda-unzip	126.8	14.15	1.05
csv-loader	86.3	42.9	3.45
anishsana	15.45	0.5	0.05
load-historic-data	119.5	7.05	1.6
nikoshen*	218.1	702.55	0
s3-java	485.5	0.50	0
upload-survey*	88	2350.55	0

5) `csv-loader` [29]: when an archived CSV file is inserted/updated into an `S3` bucket, the lambda it extracts the CSV file; and creates a DynamoDB entry for each record within the CSV file.

6) `nikoshen` [30]: when an object is inserted/updated into an `S3` bucket, the lambda it checks whether a matching DynmoDB item exists. If not, it creates it.

7) `upload-survey` [31]: when an object is inserted/updated into an `S3` bucket, the lambda serializes the received data, and updates the DynamoDB with their information.

-Implementation: We used Zest [10, 32], the state-of-the-art Java fuzzing tool to add our extension. Zest is a GBF that allows users to write input generators. To compute coverage, Zest instruments dynamically loaded classes to collect coverage information. We built our extension on top of Zest; we call our tool **SpotOn**. Our implementation utilizes an experimental codebase within Zest that defines FCI as a set of execution indices with values. We extended this work to amend the types of the generator's call stack to each execution index object of the FCI. We managed child generation by implementing a heuristic in both tools, ensuring inputs with significant time aren't overrun with excessive children. As processing time for children increases, the number of tests decreases. This heuristic embodies $\textsc{numCandidates}(i_c)$ in Alg. 1. Additionally, we utilized the Wala [17] framework for static analysis, running it offline before executing SpotOn. Lastly, in SpotOn, mutation alternates between targeted and random mutations with a 50% probability.

-Generators: To fuzz AWSLambda applications, we wrote an AWS generator package, which populates AWS resources interacting with the AWSLambda application, such as S3 and DynamoDB instances. We assume that the structure of the DynamoDB table is known; i.e., its primary key and range key; if one exists. Before every test, the entire AWS state is cleared and repopulated. We randomly generate files of various types and add them to an S3 bucket. Supported types include XML, image, CSV, text, Java class, JavaScript, and archive files. Some of these formats already existed in Zest, others were created by the authors of this paper. These generators have a 0.003 probability of creating erroneous files, like missing or incorrect file extensions. In contrast, for DynamoDB, we randomly generate items with various column sizes (excluding primary and range keys) and populate the resource with them.

-Setup: We used *LocalStack 1.4.0* [33]: a local cloud software development framework used to develop, test and run AWS applications locally. We ran our experiment over a Dell Inc. Precision T3600 machine with 32 GB RAM running Ubuntu 22.04.4 LTS. In our evaluation, we utilized Zest's Maven plugin, JQF, which instruments coverage code within dynamically loaded classes for fuzzing. Coverage of already loaded classes (e.g., java.lang, etc.) is not included in the results of this section. Additionally, coverage for infrastructure applications like com.amazonaws and com.ibm.wala was excluded, as it is beyond the scope of this research.

4.1 RQ1: Does SpotOn achieve more coverage?

To answer this question, we fuzzed each benchmark using both Zest and SpotOn, for an hour with 20 repetitions. We categorize coverage results into two categories. (1) Application code coverage: defined as code targets (branches) that exist within the parent package of the program under test. (2) All code coverage: defined as all code targets including those in third-party libraries.

Fig. 2 shows results for the application coverage, where *Common* lists the common code targets found by both tools, while additional ones found by SpotOn, and Zest are listed in +*SpotOn*, and +*Zest*, respectively. Fig. 3 shows results for all-code coverage. We notice that in both tables, most of the code targets were found by both tools, though SpotOn on average finds more code targets. This is mostly apparent in `nikoshen` and `upload-survey`. In `nikoshen` type-targeting was useful to focus the mutation on the DynamoDB Item to create a matching and a valid item. In `upload-survey` type-targeting was useful to focus on the name of the S3 object. Both benchmarks used the constant string lookup to generate specific string constants. In other benchmarks, we notice that type-targeting is useful for more stable results (more in the following section).

Finally, observe the correlation between application coverage and overall code coverage. Achieving higher application coverage generally leads to increased overall coverage. This is because covering a new branch in the application code often triggers the execution of additional code from third-party libraries. Thus, prioritizing application coverage is likely more efficient and leads to higher overall code coverage

4.2 RQ2: What is the effectiveness of type-based targeting?

To isolate the effect of the type-based targeted mutation form the constant string optimization, we separated the later into a separate mode: ZestStrOpt. Fig. 5 through Fig. 11 shows the overall coverage for Zest, ZestStrOpt, and SpotOn. The solid line in the graph describes the median for all 20 repetitions of the experiment. While the shaded lines describe the 95% confidence interval.

The first observation is that in all benchmarks, SpotOn performed either the same or better than Zest. In the case of `s3-java`, neither ZestStrOpt nor SpotOn demonstrate significant coverage improvements over the baseline Zest. However, we believe this outcome inaccurately represents the effectiveness of type-based targeting, primarily due to a limitation in collecting coverage from dynamically loaded libraries. Particularly in this benchmark, the new coverage achieved by type-based mutation stemmed from pre-loaded classes, which were consequently omitted from the overall coverage collection. In `lambda-unzip` 6, `load-historic-data` 7, and `anishsana` 8, we do not see a significant coverage difference. However, we can observe that SpotOn results are more reliable among the three tools in `load-historic-data` and `anishsana`. This is apparent due to its relatively smaller confidence interval. Similarly, we can see stable results for SpotOn in `nikoshen`, `csv-loader` and `csv-loader`.

We see significantly better performance for SpotOn and ZestStrOpt in `nikoshen` 9, and in `csv-loader` 10. In fact, the median of the baseline Zest is at the very bottom suggesting that most of the runs were performing poorly compared to SpotOn and ZestStrOpt. The orange shading in `csv-loader` suggests that a run or two from the population, Zest was able to craft the right input that achieved the desired coverage. In `nikoshen` the string constant was needed to create a matching state between the DynamoDB item and the S3 object. In `csv-loader` the string constant was needed to create an S3 object with a particular name. In both benchmarks, the type-based targeted mutation was useful to speed up the coverage by focusing the mutation on the substructure with the matching influencing type. In general, we can conclude that SpotOn has the best reliable performance in coverage finding among the three extensions.

4.3 RQ3: What is the overhead of SpotOn?

To identify the performance bottleneck for SpotOn, we divided the fuzzing operations into 4 processes: (1) mutation: the process of mutating subsequences of the FCI, (2) generation: the process of constructing an input by running the input generators on the FCI, (3) testing: the process of running the program under test with the input generated from the generation step, and (4) handling: the process of handling test result, i.e., new coverage, success or failure.

Our primary experiment's data revealed that the combined impact of mutation and handling processes accounts for no more than 3% of the total execution time. To assess the effect of the generation and testing steps, we conducted a controlled experiment in which SpotOn managed two distinct FCI types: a linear FCI to mimic Zest's operations and an EI-enabled type-annotated FCI. We utilized the linear FCI for the generation phase and the annotated FCI for the mutation process. This setup ensures that both Zest and SpotOn produce the same inputs, but SpotOn additionally creates and maintains the annotated FCI. We ran both tools using this configuration for a fixed number of tests. For each benchmark, we used three-quarters of the tests from the one-hour time-out experiment. The following results report the averages of the 10 runs.

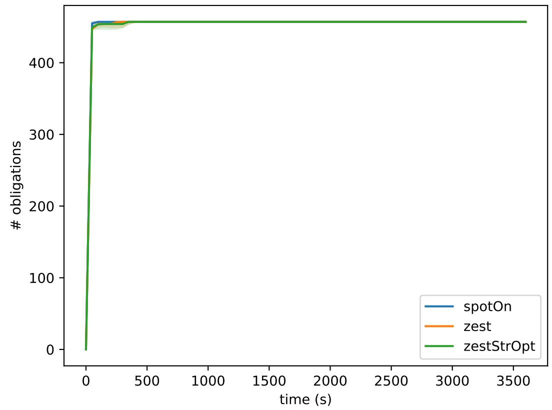

Figure 5: Median coverage of s3-java

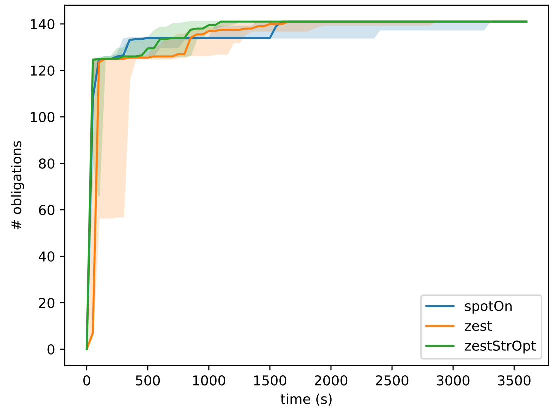

Figure 6: Median coverage of lambda-unzip

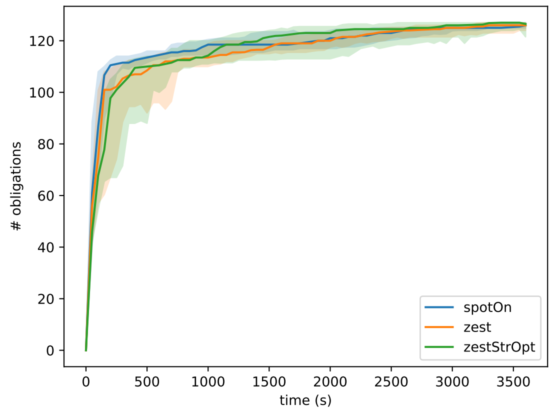

Figure 7: Median coverage of load-historic-data

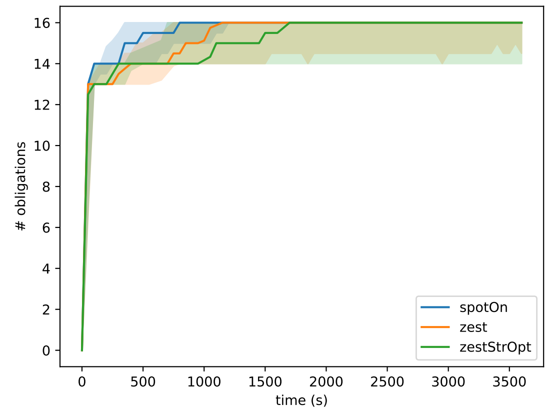

Figure 8: Median coverage of anishsana

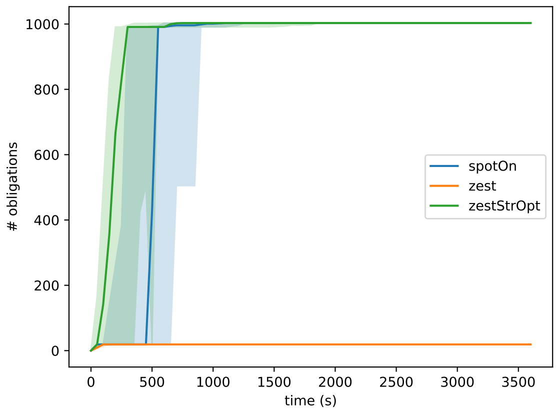

Figure 9: Median coverage of nikoshen

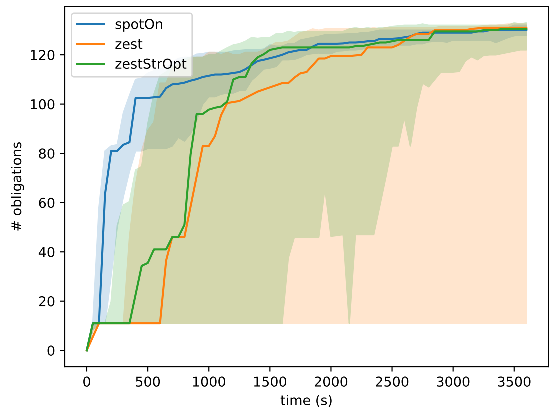

Figure 10: Median coverage csv-loader

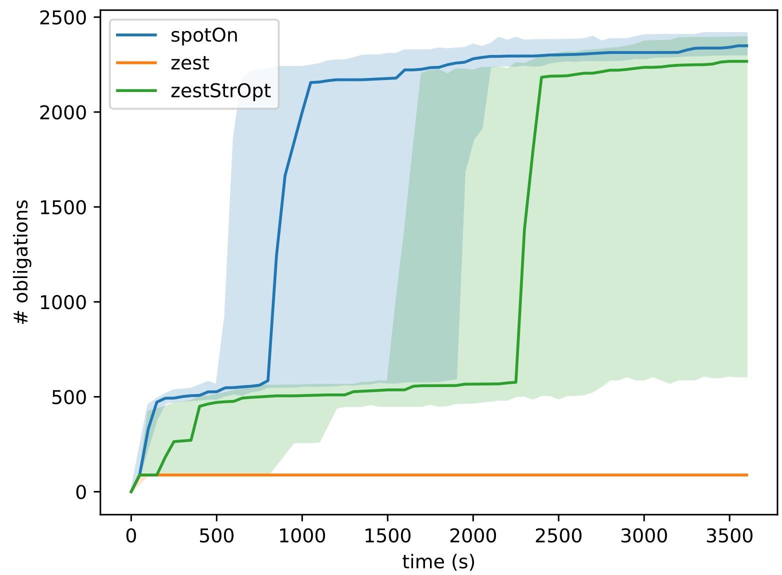

Figure 11: Median coverage of upload-survey

Fig. 12 shows the average generation time per test. We notice that the average generation cost of SpotOn is not too far off from Zest's. This overhead is due to computing and maintaining the annotated FCI. More precisely, during the generation SpotOn intercept, every method invocation to keep track of EI counters used in their creation. Then, when a `java.util.Random` method is about to execute, SpotOn creates the EI instance and looks up its corresponding value within the FCI. If found, the generation uses the present value, otherwise a fresh value is created and the triple of the EI object, its annotated type, the fresh value is added to the FCI.

Also, we measured the usage of the Java heap during experiments. We noticed that both Zest and SpotOn have a high rate of heap allocations-deallocations forming a sawtooth-like shape for heap usage over time. Overall, we observed that SpotOn's memory overhead ranges from a minimum of 0.82x to a maximum of 4.62x, with an average overhead of 3.29x when compared to Zest's.

5 Related Work

Coverage-guided fuzzing (CGF) [1, 34, 2, 3, 4, 5, 6] is a random testing technique with lightweight instrumentation for coverage computation. Using the coverage information, fuzzing tools can then make informed decisions about the choice of interesting input likely to achieve new coverage if mutated. CGF revealed many bugs in widely used/tested programs, such as in Clang, OpenSSH, JavaScriptCore, LibreOffice, Python, SQLite, Google closure-compiler, JDK, Mozilla, and BCEL to mention a few [1, 35, 36].

However, as CGF lacks information about the input structure, its effectiveness can be negatively affected. Grammar-based fuzzers such as CSmith [37], jsfunfuzz [38], and Grammarinator [39] use a declarative form of the input that defines input's grammar. Some works used the existence of a test corpus to craft new inputs from existing tests [40, 41, 42, 43]. Most of this work relied on understanding grammers and/or features of existing corpus and then use structural mutation to generate new inputs.

Generator-based fuzzers (GBF) utilize programmatically defined generators to create structured inputs. We view grammar-based fuzzers and generator-based fuzzers as orthogonal techniques, with the latter naturally imposing a language-type structure on the input. This structural information can be leveraged during input generation, the primary focus of this paper. More recently, Padhye el. [10] introduced generator-based fuzzers by extending generator-based testing [9] to use random input generators. In our work, we extend Padhye el.'s work to enable type-based targeted mutation.

In CGF, obtaining useful seeds has been an active area of research [2, 3, 16, 44], where some work focused on developing heuristics for picking seeds, and others focused on finding values to create good seeds. For example, Marcel Bohme et al.[2] used a Markov chain to assign energy to seeds, prioritizing inputs that exercise low-frequency paths for

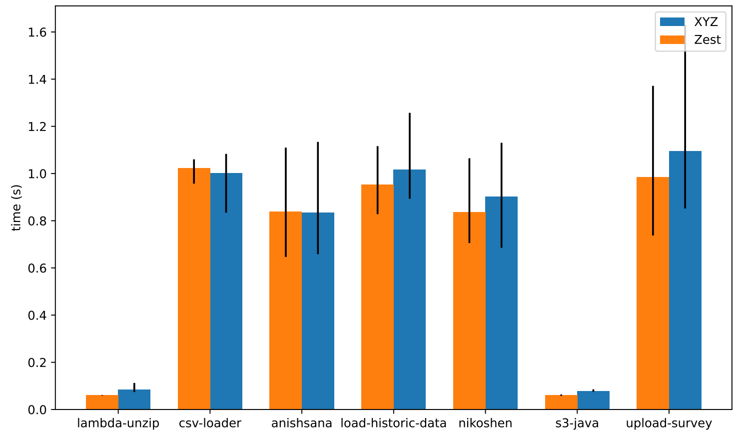

Figure 12: Average generation cost per test; errorbars between 25th and 75th percentiles

better coverage. AFLGo[3] computes the closeness of a seed to targeted code locations and gradually assigns more energy to seeds closer to the target. While these approaches focus on input selection heuristics, our work concentrates on mutating inputs based on their types. μ^2[16] controlled seed selection by computing a mutation score for each seed, while Pythia[45] used a learning-based technique to identify useful mutations. FairFuzz [46] computes rare test targets and freezes parts of seeds that contribute to reaching these targets, mutating the remaining parts. Other tools[47, 12, 13] use a form of symbolic execution to compute useful values for the input. The key advantage of this technique is that, unlike the heuristic-based approach, symbolic execution can turn the search for new inputs into a constraint set. When these constraints are successfully solved, they yield a new input that effectively covers the intended targets. The cost, however, is that these tools are usually slow due to the overhead of constructing and solving constraints.

The work we present in this paper falls into the first category of heuristics for picking seeds, which tries to craft a likely useful input using type-based targeted mutation. In contrast with FairFuzz, our work finds the places where mutations are needed, rather than the places that should be avoided. Our intuition is that for any given branch in the program, there are usually a few places in the input that are relevant to reaching the intended branch.

6 Conclusion and Future Work

In this paper, we introduced a novel type-based mutation heuristic with constant string lookup for generator-based fuzzers. Results show improvements in application coverage of an average of 18.2% (geometric mean) over the baseline, and improvements in coverage including third-party code of 43.2% when the set of covered libraries remained the same, and even more so when our approach covers additional libraries.

An essential aspect of our technique involves the composition of generators directly impacting coverage results. In the future, we plan to investigate the automation of their creation. Also, since type-base targeting can be limited when for example, the input is an array, we plan to leverage the object dynamic information to target particular objects of the same type.

7 Data Availability

Code and experiment reproducibility instructions will be made public once the paper is accepted. An anonymized preliminary version is available for review [48].

References

[1] Michal Zalewski. American fuzzy lop (afl). `https://lcamtuf.coredump.cx/afl/`, 2014.

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[4] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[5] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[6] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[7] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[8] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. Enhancing coverage-guided fuzzing via phantom program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1037–1049, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[10] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009.

[12] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[13] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. Confetti: Amplifying concolic guidance for fuzzers. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 438–450, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 110–120, New York, NY, USA, 2009. Association for Computing Machinery.

[15] OpenAI. Chatgpt 3.5. `https://chat.openai.com`.

[16] Isabella Laybourn. μ_2: Using mutation analysis to guide mutation-based fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 331–333, 2022.

[17] Joanna C. S. Santos and Julian Dolby. Program analysis using wala (tutorial). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1819, New York, NY, USA, 2022. Association for Computing Machinery.

[18] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, 1999.

[19] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 238–248, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Christopher S. Meiklejohn, Rohan Padhye, and Heather Miller. Distributed execution indexing, 2022.

[21] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 110–120, New York, NY, USA, 2009. Association for Computing Machinery.

[23] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 141–152, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Rohan Padhye. Jqf. `https://github.com/rohanpadhye/JQF`.

[25] awsdocs. S3-java. `https://github.com/awsdocs/aws-lambda-developer-guide`, 2017.

[26] anishsana. anishsana. `https://github.com/anishsana/serverless`, 2017.

[27] rmalired. load-historic-data. `https://github.com/rmalired/serverless`.

[28] Craftware. lambda-unzip. `https://github.com/Craftware/aws-lambda-unzip`, 2013.

[29] pauldeng. csv-loader. `https://github.com/pauldeng/aws-s3-lambda-dynamodb-csv-loader`, 2013.

[30] nikosheng. nikoshen. `https://github.com/nikosheng/aws-serverless-lambda`, 2018.

[31] tony schndr. upload-survey. `https://github.com/tony-schndr/S3UploadSurveyToDynamoDB`.

[32] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[33] LocalStack. Localstack. `https://localstack.cloud/`. [Online; accessed 14-May-2023].

[34] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[35] LibFuzzer. Libfuzzer trophies. `https://llvm.org/docs/LibFuzzer.html#trophies`.

[36] Rohan Padhye. Jqf github. `https://github.com/rohanpadhye/JQF`.

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[38] W. Syndder and M. Shaver. jsfunfuzz. `https://github.com/MozillaSecurity/funfuzz?tab=readme-ov-file`.

[39] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 45–48, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 724–735. IEEE Press, 2019.

[41] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017.

[42] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 38, USA, 2012. USENIX Association.

[43] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, 2020.

[44] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing a test corpus with bonsai fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 723–735. IEEE, 2021.

[45] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations, 05 2020.

[46] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008.

[48] Anonymized. Anonymized github repo for xyz. `https://anonymous.4open.science/r/Artifcat-6F20/`, 2024.