PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems

Wen Li, Jinyang Ruan, and Guangbei Yi, *Washington State University;*
Long Cheng, *Clemson University;* Xiapu Luo, *The Hong Kong Polytechnic University;*
Haipeng Cai, *Washington State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/li-wen

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the
32nd USENIX Security Symposium
is sponsored by USENIX.

PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems

Wen Li[1], Jinyang Ruan[1], Guangbei Yi[1], Long Cheng[2], Xiapu Luo[3], and Haipeng Cai[1*]

[1]*Washington State University* [2]*Clemson University* [3]*The Hong Kong Polytechnic University*

{li.wen,jinyang.ruan,guangbei.yi,haipeng.cai}@wsu.edu

lcheng2@clemson.edu csxluo@comp.polyu.edu.hk

Abstract

While offering many advantages during software process, the practice of using multiple programming languages in constructing one software system also introduces additional security vulnerabilities in the resulting code. As this practice becomes increasingly prevalent, securing multi-language systems is of pressing criticality. Fuzzing has been a powerful security testing technique, yet existing fuzzers are commonly limited to single-language software.

In this paper, we present PolyFuzz, a greybox fuzzer that holistically fuzzes a given multi-language system through cross-language coverage feedback and explicit modeling of the semantic relationships between (various segments of) program inputs and branch predicates across languages. PolyFuzz is extensible for supporting multilingual code written in different language combinations and has been implemented for C, Python, Java, and their combinations. We evaluated PolyFuzz versus state-of-the-art single-language fuzzers for these languages as baselines against 15 real-world multi-language systems and 15 single-language benchmarks. PolyFuzz achieved 25.3–52.3% higher code coverage and found 1–10 more bugs than the baselines against the multilingual programs, and even 10-20% higher coverage against the single-language benchmarks. In total, PolyFuzz has enabled the discovery of 12 previously unknown multilingual vulnerabilities and 2 single-language ones, with 5 CVEs assigned. Our results show great promises of PolyFuzz for cross-language fuzzing, while justifying the strong need for holistic fuzzing against trivially applying single-language fuzzers to multi-language software.

1 Introduction

Constructing one software system using multiple programming languages at the same time (i.e., *multi-language construction*) enables combining the best of different languages (e.g., efficiency of C and programmability of Python) hence brings many benefits (e.g., greater productivity of development process, higher performance of resulting software) simultaneously. In fact, multi-language construction has long been a normal real-world software practice and sustained its growing momentum for decades [39]. Yet this practice also brings additional threats to cybersecurity. That the use of multiple languages introduces more security vulnerabilities in the resulting multilingual code [24] is not just a statistical finding—recent work [40] has demonstrated the prevalence and criticality of those vulnerabilities (e.g., CVE-2021-41497, CVE-2021-41500, and 6 other CVEs, all with high severity scores), mainly induced by *cross-language* information flow.

Statically analyzing the information flow would suffer an excessive rate of false positives, in addition to the questionable feasibility of doing so—such analyses would be heavily language-specific hence hardly extensible to support diverse (e.g., heterogeneous semantics of different languages, variations in cross-language interfacing mechanisms) multilingual code. Dynamic information flow analysis [40, 73] largely overcomes these limitations, but its vulnerability discovering capabilities are bounded by the typically quite limited coverage of available test inputs. This weakness could be further mitigated by test input generation techniques like fuzzing, which in fact has been the de facto standard technique for software vulnerability discovery [46].

However, existing fuzzing techniques (e.g., [11, 16, 20, 22, 41, 65]) are exclusively aimed at single-language software and predominantly focused on C/C++, including recent ones [13, 42] that seemingly fuzz across different *language unit*s (i.e., code written in one language) but actually one language unit still. We may trivially apply these single-language fuzzers to multilingual code (e.g., by simply fuzzing the *entry* language unit). Yet that would essentially treat other language units as black boxes in entirety, dismissing cross-language interactions hence largely compromising the potential of fuzzing.

In this paper, we propose to holistically fuzz multilingual code to empower systematical vulnerability discovery in real-world multi-language systems. To strike a good balance between scalability and effectiveness, we focus on greybox fuzzing as most prior peer works did [46]. In light of the aforementioned limitations of extant solutions, we aim to (1) offer significantly greater cost-effectiveness (i.e., achieving higher code coverage and finding more bugs within a given amount of time) and (2) offer practical extensibility to support multilingual code with different language combinations and interfacing mechanisms. Fulfilling both aims would justify

*Haipeng Cai is the corresponding author.

the need of *holistic* fuzzing for multi-language software, but it also faces two major challenges accordingly.

First (*Challenge-1*), earlier findings revealed that the additional vulnerabilities of multilingual code (beyond those within each language unit) are a result of cross-language information flow [40]. With greybox fuzzing, we do not want to explicitly analyze language interfacing (which would not only compromise fuzzing efficiency but also impede language extensibility). However, there is no prior knowledge on how to generate inputs that effectively exercise information flow across heterogeneous language units with a random testing technique like greybox fuzzing.

Second (*Challenge-2*), it is known that multi-language software can be highly diverse, using a large variety of languages combined [39]. Developing a separate fuzzer dedicated to each particular language combination is clearly undesirable and may not even be feasible. Meanwhile, greybox fuzzing does require some knowledge about program internals—thus, downgrading to blackbox fuzzing hence achieving trivial extensibility is not an option. However, acquiring such knowledge necessitates language-specific analyses, which potentially compromises the extensibility.

To address these challenges, we developed PolyFuzz, a novel greybox fuzzer that achieves holistic fuzzing of multilingual code. PolyFuzz realizes whole-system code coverage measurement and feedback to guide seed scheduling in a systematic fashion. More importantly, it starts with a *seed generation* phase to overcome the common scarcity of initial seeds (especially in multi-language systems as we found). During this phase, PolyFuzz exploits a sensitivity analysis to explicitly model the semantic relationships between (various segments of) inputs and branch predicates based on regression. It then proceeds to conventional fuzzing and adaptively switches back to seed generation when necessary. In this way, PolyFuzz achieves holistic fuzzing while being aware of cross-language information flow, hence addressing *Challenge-1*. To maximally support different language combinations, PolyFuzz employs a minimal language-specific analysis for holistic coverage measurement and harvesting only the variable values necessary for learning the regression model. This is enabled by a custom intermediate representation (IR) that unifies run-time value probing across heterogeneous languages, which makes the rest (and most) of PolyFuzz language-agnostic, addressing *Challenge-2*.

We have implemented PolyFuzz based on AFL++ [16] and applied it to 15 real-world multi-language systems, including both Java-C and Python-C benchmarks to demonstrate its extensibility to support different language combinations. Without existing multi-language fuzzers available, we compare PolyFuzz against three state-of-the-art single-language fuzzers (Honggfuzz [65] for C, Jazzer [11] for Java, and Atheris [22] for Python) as baselines. Our results show that, with the same 24-hour time budget and same initial seeds, PolyFuzz achieved 25.3% and 52.3% higher block coverage and found 1, 10 more bugs than Jazzer and Atheris, respectively, which justifies the necessity of *holistic* fuzzing and insufficiency of trivially applying single-language fuzzing against multilingual code. Notably, PolyFuzz enabled discovery of 12 new multilingual vulnerabilities and 2 new single-language vulnerabilities with 5 CVEs assigned, of which 2 have been fixed by developers by the time of paper writing. We also demonstrated comparable or even greater cost-effectiveness of PolyFuzz over the three baseline fuzzers against commonly used single-language benchmarks, and validated the significant contribution of its sensitivity-analysis-based seed generation module to its overall performance superiority. On these single-language benchmarks, PolyFuzz achieved 11.0%, 20.1% and 10.1% higher block coverage than Jazzer, Atheris, and Honggfuzz, respectively; PolyFuzz also achieved 7.6% and 11.4% higher block and path coverage, respectively, than AFL++.

To the best of our knowledge, PolyFuzz is the first holistic multi-language fuzzer. Its open-source, extensible design also facilitates the development of greybox fuzzing of other language combinations beyond those among Java, C, and Python. Importantly, we note that the lack of multilingual fuzzing benchmarks was a tremendous barrier to our evaluation—in contrast, there are standard single-language fuzzing benchmarks available for existing fuzzers to use. Thus, we also contribute to the community with the first benchmark suite for multilingual fuzzing. The PolyFuzz source code and this suite have been made available at Figshare.

2 Background and Motivation

In this section, we give a brief background of greybox fuzzing and discuss various challenges to fuzzing multi-language software, hence motivating our work with a real-world example.

2.1 Greybox Fuzzing

Greybox fuzzing [6, 16, 46, 49, 61] perform lightweight static or dynamic analysis on the targets and/or gather execution feedback (e.g., coverage) to guide seed selection and/or mutation [20, 45]. The general workflow of greybox fuzzing is a loop as follows. The fuzzer (1) maintains a seed queue Q, which can be updated during fuzzing; (2) selects some seeds from Q following a certain policy; (3) mutates the seeds in various ways (e.g., bit/byte flips, simple arithmetics, stacked tweaks and splicing [45]); (4) run the target program with the newly generated test cases, and reports vulnerabilities or updates Q if necessary; and (5) goes back to step (2).

Although greybox fuzzing has become quite popular with its high efficiency, the latest research indicates that more than 91.7% executions in the state-of-art fuzzing process are redundant due to the unreachable inputs [79]. Various techniques, such as data-flow-sensitive fuzzing and deep-learning guided mutation, have been proposed to remedy the problem.

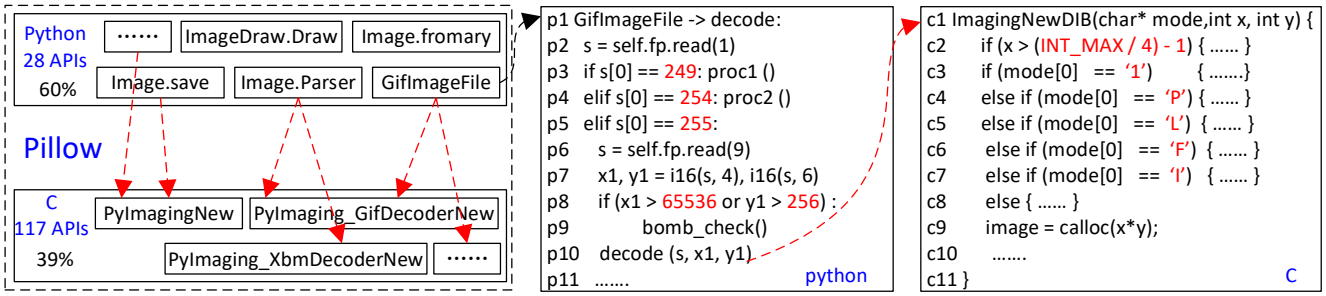

Figure 1: Motivating example: a real-world multilingual program Pillow.

2.2 Fuzzing Multi-Language Systems

Greybox fuzzing has demonstrated high effectiveness in exposing vulnerabilities in real-world programs [46]. Excellent fuzzers such as AFL [49] and libFuzzer [43] have succeeded in detecting more than 16K vulnerabilities in various projects [45]. However, to the best of our knowledge, state-of-the-art fuzzers target single-language code [46]; while in modern software development, the status is that 80%+ of the studied projects are programmed in multiple languages [1]. Applying these single-language fuzzers to multilingual code suffers various limitations as follows, among others:

- *Feasibility for different languages.* In general, the interfaces between different languages in multi-language software are complex and diverse. Thus, run-time input formats vary across language units and APIs. As a result, it is not always feasible in practice to construct proper calling contexts and fuzzing instances for all APIs. As an example shown in Figure 1, after analyzing 28 Python APIs and 117 C APIs in Pillow [56], we found that the input format and calling context of these APIs are diverse. Thus, to fuzz the language units separately in Pillow, we need to develop 28 fuzzing drivers for the Python unit and 117 for C. These drivers are expensive to develop and maintain, and require substantial computing resources to run them.
- *Inefficiency due to incomplete feedback.* When fuzzing a multilingual program, single-language fuzzers can fail to evolve the fuzzing process due to the lack of holistic coverage feedback. For example, when fuzzing Pillow of Figure 1, a Python fuzzer would treat the C units (which account for 39% of the system in code size) as black boxes, failing to perceive the coverage changes in these units. Without the holistic feedback, the fuzzer suffers inefficiency.
- *Reproducibility of vulnerabilities.* Simply fuzzing language units separately may lead to semantic loss between units due to the looser constraints than the whole system execution. For example, as shown in Figure 1, when a C fuzzer fuzzes the API *ImagingNewDIB*, since only variable x is validated at line c2, an `Out of Memory` (OOM) can happen at line c9 if the value of $x \times y$ is large enough during fuzzing mutation. However, this report is a false positive since a bomb check exists at line p9 in Python on the complete data flow path. Hence, vulnerabilities detected by such fuzzers may fail to be triggered in actual executions.

These drawbacks of single-language fuzzers motivate us to develop a cross-language fuzzing technique, achieving holistic, whole-system fuzzing (WSF). However, although WSF can solve the problems discussed above, it faces another inefficiency challenge. As shown in Figure 1, compared with either a Python or C fuzzer, the WSF fuzzer spends more time on executions due to the whole-system instrumentation (both Python and C units); moreover, per our experience, the scarcity of initial seeds (for exercising *cross-language* behaviors) is a *peculiar* barrier to multilingual greybox fuzzing, albeit quality seeds are essential for both multilingual and single-language fuzzing. Prior work shows that random mutation guided by control flow coverage causes over 91.7% redundant inputs [79]—this ratio was up to 95% in our experiments with multilingual fuzzing due to the much higher code complexity. For the code snippets in Figure 1, three variables (i.e., s[0], x1 and y1) control all the branches in Python and two variables (i.e., mode[0] and x) in C. These variables are reachable during the runtime executions, but it is hard to mutate them to expected values (e.g., let input[0]=249) randomly. To hit or reverse these branches, WSF needs more precise guidance to mutate specific positions (*where*) in the inputs. In this example, we may first identify that s[0] is the first byte of input, and x1 is extracted from the 1st to 4th byte. Then, we can mutate these positions into specific values (*how*) (e.g., let input[0]=249,254 or 255) to cover all the blocks quickly. Some form of cross-language information flow analysis is necessary to support this kind of precise mutation.

Based on all the observations above, we propose a cross-language fuzzing framework PolyFuzz to enable efficient, holistic fuzzing of multi-language systems.

3 The PolyFuzz Framework

In this section, we present the design of our multilingual fuzzing framework, starting with an overview (§3.1) of PolyFuzz followed by the details of its three main mod-

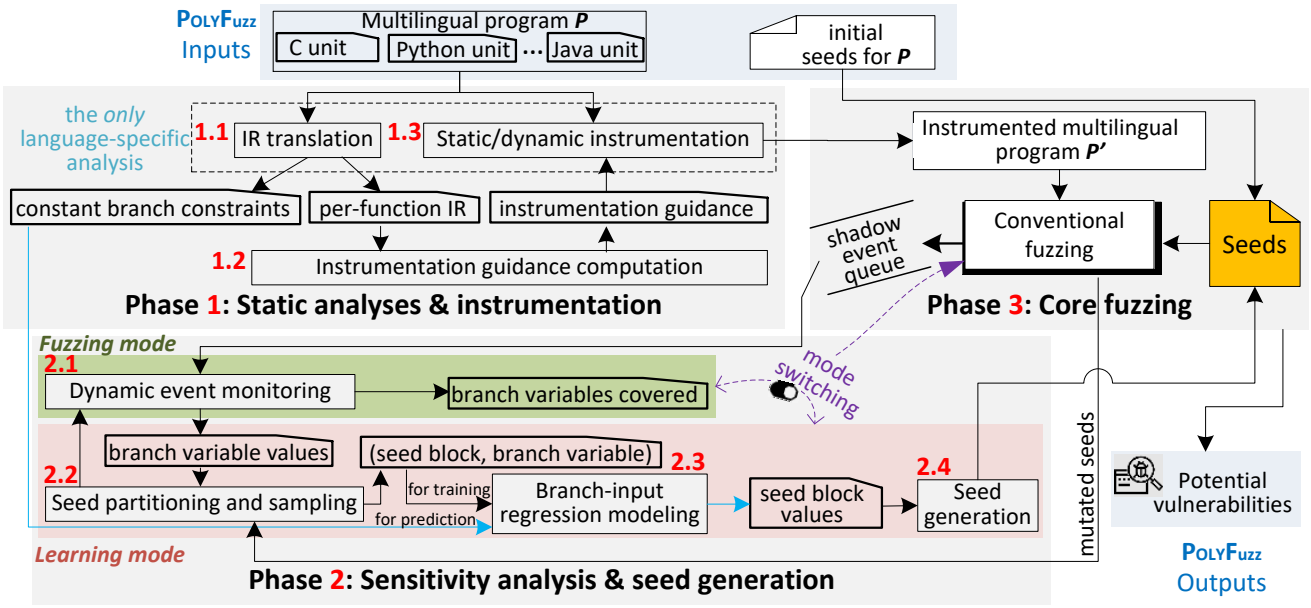

Figure 2: An overview of PolyFuzz's architecture, including its inputs, three working phases and per-phase steps, and outputs.

ules/phases: *static analysis&instrumentation* (§3.2), *sensitivity analysis&seed generation* (§3.3), and *core fuzzing* (§3.4).

3.1 Framework Overview

As depicted in Figure 2, the framework takes two **PolyFuzz Inputs**: (1) the multilingual program P under testing, including various language units, and (2) the set of existing test inputs for P used as the initial seeds for fuzzing.

With these inputs, PolyFuzz works in three key phases. In **Phase 1**, it translates each language unit of P to a custom, language-independent intermediate representation for each function (per-function IR). During this translation, constant branch constraints (i.e., constants in a branch predicate) are also extracted to support seed generation later on. Based on such per-function IRs, basic blocks and branches therein if any (together noted as *instrumentation guidance*) to probe for are computed to guide the instrumentation of P. The main goals of this phase are to (1) enable whole-system coverage measurement hence lay the basis for holistic fuzzing, while minimizing the probing scope hence instrumentation-induced overhead, and (2) minimize language-specific analysis hence maximize the language extensibility of multilingual fuzzing. As marked, only the IR translation and instrumentation are *language-specific*, making *the rest of the framework language-independent*—the purpose of the IR and the translation.

With the instrumentation program P', PolyFuzz focuses on generating more seed inputs in **Phase 2**, working between two modes. It starts in the **Fuzzing mode**, in which it informs the core fuzzing (**Phase 3**) to run P' against the initial seeds. Meanwhile, it monitors the *dynamic events* (i.e., coverage info and values of covered branch variables) that are pushed to a shadow-memory based queue (noted as *shadow event queue*) by probes inserted in **Phase 1**. From these events, if it finds newly covered branch variables, it pauses the core fuzzing and switches to Learning mode. In this mode, PolyFuzz learns a regression model that captures the semantic relationship between branches and inputs via a sensitivity analysis. This is done by retrieving the mutated seeds (from the core fuzzing) that led to the new coverage and partitioning each seed into seed blocks (a sequence of bytes of a specified length), followed by (randomly) sampling (i.e., via mutation) possible values of each seed block. Meanwhile, it invokes the same dynamic event monitoring step to harvest branch variable values observed during the seed sampling.

Once the sampling is done, the core fuzzing is informed to resume and PolyFuzz switches back to **Fuzzing mode** while continuing in the Learning mode (in parallel). On the resulting values for each (seed block, branch variable) pair, a regression model is trained. At the model inference time, new branch variable values are sampled (to cover both outcomes of each related branch) by expanding the related constant branch constraints, and used as model test inputs to predict respective seed block values. These predicted values are then used to generate new seeds, which are added to the seed corpus feeding the core fuzzing. Any seeds covering new branch variables during the (parallel) **Fuzzing mode** while the current Learning mode is ongoing are cached for later/queued processing.

In **Phase 3** (core fuzzing), the fuzzer core of PolyFuzz runs a path-coverage guided conventional fuzzing algorithm. When triggered, potential vulnerabilities, along with the triggering seeds, are reported as **PolyFuzz Outputs** for bug confirmation and reproduction.

3.2 Static Analysis & Instrumentation (Phase 1)

To fulfill its two main goals (§3.1), this phase works in three technical steps as described in the following three subsections.

3.2.1 IR Translation (Step 1.1)

Per its overall workings, POLYFUZZ requires sufficient but minimal probing for (1) the coverage of basic blocks hence that of distinct program paths, as the basis of holistic fuzzing, and (2) the definitions of branch variables, as needed for the sensitivity analysis. To that end, both control and data flow analyses are necessary, which are heavily specific to different languages hence compromising the framework's extensibility to support other languages. To overcome this challenge, we propose a new custom IR to enable unified (language-independent) analyses to meet both probing requirements. Since it is particularly needed for making the sensitivity analysis (SA) extensible, we refer to the IR as *SAIR*.

SAIR Definition. Unlike a typical IR (e.g., as used by a compiler), SAIR is an IR specialized for *only capturing the most essential information for fuzzing*, rather than representing the entire program. As per the two requirements above, SAIR focuses just on basic blocks and the definitions of branch variables. Accordingly, the formal syntax of SAIR is:

$$
\begin{array}{lll}
P & ::= & F^* \\
F & ::= & \tau f(x^*)S^* \\
S & ::= & [x =]e^* \mid [cmp]e^*, e^* \\
e & ::= & \tau x \mid C \mid \varepsilon \\
\tau & ::= & I \mid O
\end{array}
$$

A program P is a sequence F^* of function definitions. A *function* F has the return type τ, function name f, a sequence x^* of parameters, and a sequence S^* of statements. The return type τ of f is one of two kinds: integer (I) and other (O)—because our current sensitivity analysis only fits regression models for integer variables; learning such models for other types of values are left for future work. Thus, we only differentiate integer or not as value types. A statement S is one of two kinds: *line* ($[x =]e^*$) formulates all non-comparison (e.g., assignment, call, and return) statements; and *cmp* ($[cmp]e^*, e^*$) defines a predicate (i.e., the comparison between two variables). An expression e is one of three kinds: a variable x with type τ, a constant C, and ε (empty string). All our control/data flow analyses based on SAIR are intraprocedural. Thus, SAIR treats all *line* statements as assignments.

Translation. Based on the definition, a language unit is translated to its SAIR via simple syntactic parsing of the unit, one function at a time, as outlined in Algorithm 1. For a given function, it first translates the declaration (line 2), followed by traversing all of its basic blocks (lines 4-15). For each basic block, the translator records the information of all its ancestors and descendants for control flow graph construction. In a basic-block, the translator parses one statement after another (lines 6-15). For a predicate statement (line 9), it decodes the

Algorithm 1: Translate a given function to SAIR

```
   Input: 𝔽: a given source function
   Output: F_sair: the SAIR of 𝔽
1  Function translate2SAIR (𝔽)
2      S_sair ← getFDeclaration (𝔽);            //translate function declaration
3      F_sair.append (S_sair);
4      foreach B_i in 𝔽 do
5          B_sair ← initBlock (F_sair, B_i) ;    //initialize current basic-block
6          foreach S_i in B_i do
7              if is_cmp (S_i) then
8                  _, Uses = getDefUse (S_i);
9                  S_sair ← getCmpSAIR (Uses[0], Uses[1]);
10                 if hasIntConstant (S_i) then
11                     dumpBrVariable (S_i);//dump branch variables with int const
12             else
13                 Def, Use = getDefUse (S_i);
14                 S_sair ← getLineSAIR (Def, Use);
15         B_sair.append (S_sair);              //insert S_sair to current basic-block
16     return F_sair
```

uses and constructs a *cmp* statement in SAIR; if this predicate has an integer constant, the branch variable information (i.e., its unique identifier, operator type such as 'less than', and the constant value) is recorded (line 11). Other source statements are translated to *line* statements. Since other code constructs are not needed for either of the two probing requirements, which SAIR serves, they are dropped during the translation.

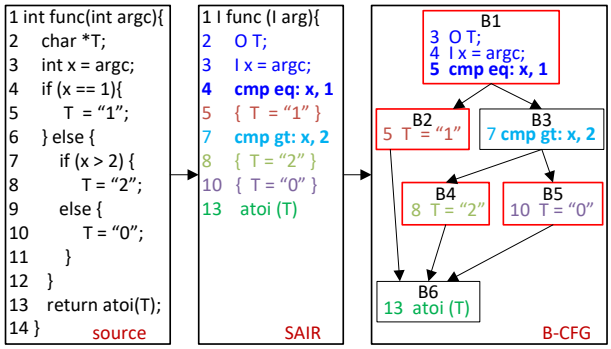

Figure 3: An illustration of translating a language unit source to its SAIR and then to the block-level CFG (B-CFG).

As an example, Figure 3 shows the translation of a source program to its SAIR, and then to its block-level control flow graph (B-CFG). At line 2, the (non-integer) type of variable T is translated to type O. At line 4, the predicate statement is translated to a *cmp* statement with the operator type *eq* and two parameters x and 1. The original return statement at line 13 is translated to a *line* statement without left value; so on and so forth. During the translation, the basic-block information is saved; hence SAIR can be easily further translated to B-CFG, as exemplified in the rightmost of Figure 3.

3.2.2 Instrumentation Guidance Computation (Step 1.2)

Based on the SAIR for each function, the next step is to compute *instrumentation guidance* (i.e., which basic blocks and branch variables should be probed for) through intraprocedural control and data flow analysis.

Algorithm 2 shows the procedure for computing minimal instrumentation sites for a given SAIR function $\mathbb{F}$. First, the

Algorithm 2: Compute instrumentation guidance

```
   Input: 𝔽: a given function in SAIR
   Output: S_instr: a set of (SAIR) statements to be instrumented
1  Function instrguideComputation (𝔽)
2      CFG ← getFCfg (𝔽);                                   //get function CFG
3      DomBB ← calDomofBB (CFG);                            //compute dominance on CFG
4      PDomBB ← calPostDomofBB (CFG);                       //compute post-dominance on CFG
5      S_BB ← ϕ;
6      foreach basic block B_i in 𝔽 do
7          if isEntry (B_i, 𝔽) then
8              S_BB.append (B_i) ;                          //the entry block should always be instrumented
9          else
10             if isFullDominator (B_i, DomBB) ||
                 isFullPostDominator (B_i, PDomBB) then
11                 continue;
12             S_BB.append (B_i) ;                          // non-(post)dominators should be instrumented
13     S_BV ← calReachability (CFG);//calculate instrumentation sites for branch variables
14     S_instr ← merge (S_BB, S_BV);
15     return S_instr
```

control flow graph (CFG) of $\mathbb{F}$ is constructed (line 2). Next, dominance and post-dominance relationships between basic blocks in the CFG are computed (lines 3-4). For each basic block (lines 6-12), whether it should be instrumented (added to S_{BB}) depends on whether it affects control-flow-path distinction. As per this rationale, the entry block of CFG must be instrumented hence added to S_{BB}. Otherwise, if a block dominates all its immediate descendants (i.e., a *full* dominator), it does not need be instrumented as it would not affect path distinction on the CFG; same if it is a full post-dominator. Then, a definition set of all branch variables (S_{BV}) is computed via a data flow (reachability) analysis [26]. Finally, S_{instr} is obtained by merging S_{BB} and S_{BV}, and returned as output.

To illustrate Algorithm 2, Figure 3 (rightmost) marks with a red boundary the blocks (S_{instr}) that should be instrumented. To compute S_{instr}, we first compute S_{BB} (lines 6-12). Specifically, (1) $B1$ is the entry block, hence $S_{BB} = [B1]$; (2) $B2$ is not a full (post-) dominator, hence $S_{BB} = [B1, B2]$; (3) $B3$ dominates both $B4$ and $B5$ (i.e., full dominator) thus it needs no instrumentation; so $S_{BB} = [B1, B2]$; (4) both $B4$ and $B5$ are not full (post-) dominators, hence $S_{BB} = [B1, B2, B4, B5]$; (5) $B6$ post-dominates all its ancestors $B2$, $B4$, $B5$ (i.e., a full post-dominator); so now $S_{BB} = [B1, B2, B4, B5]$; To validate S_{BB}, we traverse the CFG to obtain all three paths $\{B1B2B6, B1B3B4B6, B1B3B5B6\}$. Now we remove the two full (post-) dominators ($B3$, $B6$) from these paths and get $\{B1B2, B1B4, B1B5\}$, which still distinguishes the same three paths. Thus, S_{BB} is validated. Then, we compute $S_{BV} = [B1@s4]$ where $s4$ is the definition of branch variable x used in $\{B1@s5, B3@s7\}$. Lastly, by merging S_{Bv} and S_{BB}, we have $S_{instr} = [B1@s4, B2, B4, B5]$ to guide instrumentation.

3.2.3 Static/Dynamic Instrumentation (Step 1.3)

In this step, PolyFuzz instruments at the basic blocks in the instrumentation guidance and probes for values of branch variables in those blocks. This is done via static instrumentation for each compiled-language (e.g., C) unit, and dynamic instrumentation for an interpreted-language (e.g., Python) unit.

3.3 Sensitivity Analysis & Seed Generation (Phase 2)

A general challenge to greybox fuzzing lies in the lack of sufficient and quality seeds [8, 71, 77]. Our experience is that this challenge is even greater with real-world multi-language systems in the wild. PolyFuzz addresses this challenge via **Phase** 2, which works in four steps as elaborated below.

3.3.1 Dynamic Event Monitoring (Step 2.1)

Per its overall working (§3.1), **Phase** 2 starts with the **Fuzzing mode**. While in this mode, the core fuzzing runs the instrumented program P', producing dynamic events as probed in P' and placing them in the shadow event queue. Specifically, each dynamic event consists of the identifier and value of a branch variable covered during the fuzzed execution of P'. The dynamic event monitor here first fetches events from the queue and put them to a memory database; then, it determines newly covered branch variables by checking any change of the database. If a change is identified, the framework switches to Learning mode, performing the following three steps.

The rationale for this switch is as follows. As illustrated in our motivating example (§2), a branch variable can be used at different branches. When a current (mutated) seed has (newly) covered a branch variable, we want to take the opportunity to exercise as many branches that use the variable as possible by satisfying the branch constraints. The goal of the Learning mode is to find new seeds that satisfy those constraints.

3.3.2 Seed Partitioning and Sampling (Step 2.2)

To better find new seeds, we need a finer control of where and how to change the current ones (i.e., the mutated seeds that just covered new branch variables). Thus, instead of just treating a seed as one single-byte stream [20], we propose to partition it into a stream of seed blocks, each being a sequence of bytes of equal lengths. In particular, we select block sizes such that typical lengths of an integer (1, 2, 4, 8, 16-byte) are all covered. This partitioning also increases the chance of making more meaningful changes to the seed in spawning new ones. The rationale is that an (e.g., integer) branch variable may influence the branch outcome more likely via a block (of various sizes) than via a single byte of the (seed) input. Our empirical results validated this design: on average over our studied benchmarks, the dominating portion (68.3%) of the seeds generated by PolyFuzz was learned with the seed block size of 4-byte, 9.1% with 1-byte, and 22.6% with 2- or 8-byte. That is, different seed-block sizes have different impact on seed quality; thus, considering multiple common sizes is justifiable and useful.

After the seed partitioning, PolyFuzz samples the value space of each seed block (given its potentially infinite size) by (1) randomly mutating its current value, (2) executing P' against the mutated value—not via the core fuzzing, and (3) observing the values of branch variables—again via the dynamic event monitoring (**Step** 2.1). This *sensitivity analysis*

Algorithm 3: Seed partitioning and sampling

```
Input: ℙ: the instrumented program
Input: 𝕊: a seed that has triggered new coverage of branch variables
Input: 𝕃: list of preset values of block length, e,g, {1, 2, 4, 8}
Input: ℕ: the target number of samples for each seed block
Output: SBP_lists: lists of (seed block, branch variable) values, one list per SBP
1  Function seedPtSampling (ℙ, 𝕊, 𝕃, ℕ)
2      SBP_lists ← ϕ;
3      foreach L_i in 𝕃 do
4          Pos ← 0;
5          while Pos + L_i <length (𝕊) do
6              S_Bi ← 𝕊 [Pos : Pos + L_i];          //extract a block with length L_i
7              N_s ← 0;
8              while N_s <ℕ do
9                  S' ← randMutate (𝕊, S_Bi); //mutate S_Bi and spawn a new seed S'
10                 execute (ℙ, S');              //execute ℙ with new seed S'
11                 BV_list ← collectBrValues ();
12                 updateSbBvPairs (SBP_lists, S_Bi, BV_list);
13                 N_s ← N_s + 1;
14             Pos ← Pos + L_i;
15     return SBP_lists
```

Algorithm 4: Branch-input regression modeling

```
Input: PC: preset parameter combinations of candidate types of models
Input: SBP_vals: a list of values of the given SBP
Input: BV_set: a set of constraint constants for the branch variable (in the SBP)
Output: S_sb: a set of values of the seed block in the given SBP
1  Function regressionModeling (PC, SBP_vals, BV_set)
2      RM_list ← {rbf, polynomial, linear};          //initialize regression model list
3      Acc_opt ← 0;                                  //initialize the accuracy as 0
4      Rm_opt ← ϕ;
5      Train, Test ← split (SBP_vals);               //80% for training and 20% for testing/inference
6      foreach RM[i] in RM_list do
7          Rm_i, Acc_i ← getModel (PC, RM[i], Train, Test);
8          if Acc_i > Acc_opt then
9              Acc_opt ← Acc_i;
10             Rm_opt ← Rm;                          //selected the optimal RM
11     S_sb ← predict (Rm_opt, BV_set)
12     return S_sb
13 Function getModel (PC, RM[i], Train, Test)
14     Acc_i ← 0;
15     Rm_i ← ϕ;
16     foreach pc in PC[i] do
17         rm ← trainModel (RM[i], pc, Train);       //train a model for each pc
18         res ← predict (rm, Test);
19         acc ← calAccuracy (res, Test);
20         if acc > Acc_i then
21             Acc_i ← acc;
22             Rm_i ← rm
23     return Rm_i, Acc_i
```

results in a list of *(seed block, branch variable)* pair (SBP) values, which are stored in the aforementioned memory database.

Algorithm 3 shows the procedure for seed partitioning and SBP sampling. It takes 4 inputs: the instrumented program $\mathbb{P}$, a seed $\mathbb{S}$, a preset list $\mathbb{L}$ of partition lengths, and the number $\mathbb{N}$ of samples targeted per seed block. For each partition length *Li* (line 3), the seed is randomly mutated $\mathbb{N}$ times, block by block (lines 5-14). Specifically, when sampling for each block S_{Bi} (lines 9-12), a new seed $S^{'}$ is spawned by randomly mutating $\mathbb{S}$ at S_{Bi}; after executing $\mathbb{P}$ with the resulting seed $S^{'}$, all branch variables covered in the execution are collected and put into BV_{list}; then the output lists SBP_{lists} of SBP values are updated to include the (S_{Bi}, BV_{list}) values.

3.3.3 Branch-Input Regression Modeling (Step 2.3)

To generate effective new seeds that immediately feed the core fuzzing, we fit a function approximating the semantic computation between program inputs and branch variables, followed by inferring the input values that are needed for exercising the branches in both directions according to the fitted function. This is done by, for each SBP (sb_i,br_i), first training a regression model on all the values (in the current memory database) of this SBP to capture the association between br_i and sb_i. Then, new fuzzing seeds are generated from the values of sb_i predicted by the trained model, against new values of br_i sampled by expanding the constraint constants (extracted during **Phase** 1) at the branches that use br_i. Algorithm 4 shows the regression modeling process, including model training, model selection, and model prediction/inference, *for each SBP*.

The algorithm takes preset parameter combinations of candidate types of regression models (e.g., **rbf**, **linear**), values of a given SBP, and expanded constraint constants for the branch variable in the SBP. Given a branch predicate *bv* `op` *c* that uses a branch variable *bv* and a constraint constant *c*, where `op` is the operator, the constant expansion is done by sampling two *bv* values such that one satisfies the constraint (i.e., making the predicate true) and the other failing the constraint hence falsifying the predicate, according to what *op* is.

First, the list of three model types considered is initialized as RM_{list} (line 2). Next, all the current values of the SBP are split (line 5) such that 80% are used as training data (*Train*) and the remaining 20% as test data (*Test*). For each candidate type, multiple models with the preset parameter combinations are trained on *Train*. Then, the model of the best accuracy against *Test* is selected for the current type (lines 13-23). Further, an optimal model Rm_{opt} is selected also by accuracy among the three model types (line 10). Finally, new values of the SBP's seed block are predicted by Rm_{opt} against the relevant constraint constants.

Given the diversity of program behaviors, the semantic relationships between the branch variables and seed blocks may follow a single pattern. Thus, we consider the three commonly used types of regression models and for each model explore different model parameter settings, so as to learn the best (most-accurate) model particularly for each program under test. Nevertheless, the learning can still fail (e.g., when the semantic relationships cannot be captured by a regression model indeed). In such failure cases, the predicted values, and the subsequently generated new seeds, will not be fruitful.

3.3.4 Seed Generation (Step 2.4)

In the last step of **Phase** 2, PolyFuzz generates new fuzzing seeds by assembling the seed blocks that now have new values returned by the regression modeling (**Step** 2.3).

Consider a seed *SD* that consists of *n* seed blocks sb_0, sb_1, ..., sb_n each having a possibly large number *x*, *y*, ..., and *z* of values, respectively, as illustrated in Figure 4. The block without any predicted values just carries the single original seed value at that block. Then, each assembled value of *SD* can be formulated as a seed-block sequence $SD = \{sb_0[k]sb_1[m]...sb_n[o]\}$, where *k*, *m*, *o* are value indices.

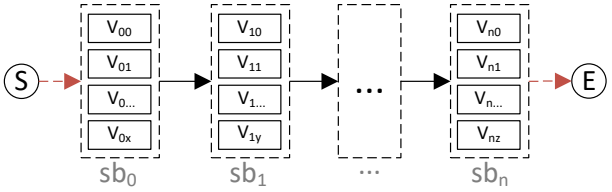

Figure 4: The graph representation of a seed.

Generally, the combinatorial space could be vast (a size of $x \times y \times ... \times z$); thus, efficiently assembling seeds is not trivial.

To address this issue, we reformulate the seed generation as a path construction problem. For a given seed, we first represent the seed blocks with values as a directed graph, where the (dummy) entry and exit nodes S and E are added for convenience (see Figure 4). Every value at a seed-block slot is a graph node, and the edges connect the blocks following the order in which they are originally located in the seed. Then, a seed-block sequence for an assembled seed value is equivalent to a graph path between S to E (with these two dummy nodes excluded). Based on these formulations, efficient seed generation works in the following two sub-steps.

Weighted sampling. Instead of exhaustively considering all possible C combinations of seed-block values, we propose weighted sampling a subset of values for each seed block, with the weight assigned as the number of branch variables covered by (any value of) the block. The rationale is straightforward—given a limited budget number M ($<C$) of combinations (i.e., generated seeds), it is desirable to prioritize the ones that led to higher coverage of branch variables (hence potentially higher branch/path coverage) by sampling more values for higher-coverage blocks. Specifically, for a seed block sb_i, the number SN_i of values to be sampled is calculated as follows:

$$(1) \quad SN_{avg} = power(M, \tfrac{1}{N'})$$

$$(2) \quad W_i = N_{bvi} \;/\; \textstyle\sum_{j=1}^{n} N_{bvj}$$

$$(3) \quad SN_i = \begin{cases} SN_{avg} + (N_{bvi} - SN_{avg}) \times W_i & N_{bvi} \neq 0 \\ 1 & N_{bvi} = 0 \end{cases}$$

where N_{bvj} denotes the number of branch variables that a seed block sb_j covers. Let N' be the number of seed blocks that cover at least one branch variable (i.e., $N_{bv} \neq 0$). First, an average sampling size SN_{avg} is calculated in (1). Then in (2), the block sb_i's weight W_i is computed as the proportion of branch variables covered by any value of this block to those covered by any value of any block. Finally in (3), SN_i is calculated by either increasing or decreasing the average (SN_{avg}) sampling size according to the weight when $N_{bvi} \neq 0$. When $N_{bvi} = 0$, as we discussed above, we assign the block with the original seed value at that block hence setting SN_i always as 1. Then, SN_i values are randomly selected for sb_i. Of course, when $M \geq C$, this sub-step is skipped.

Path construction. Now with under-sampled (when necessary) values for each seed block, seed assembling is achieved via depth-first traversal on the seed graph, by invoking the procedure of Algorithm 5 (with $d = 1$,p={}). For the current depth d, the algorithm first obtains the pre-calculated weighted sampling size (SN_d), followed by randomly sampling SN_d seed-block values (SB_V) (line 3). Then, it iterates all the values in SB_V (lines 4-9). Specifically, for each value v, it is inserted to the current path p at slot d; if the iteration reaches the exit node (at the max depth $SBL.size$), a new full path is generated and added to PL (line 7); otherwise it recursively runs the procedure to the next depth $d+1$(line 9).

Algorithm 5: Path construction for seed assembling

```
Input: SBL: the list of seed blocks to be assembled
Input: SNL: the list of weighted sampling sizes for SBL
Input: p: current path in construction
Input: d: current depth of the path p
Output: PL: the list of resulting paths (assembled seeds)
1  Function getPathByDF (SBL, SNL, p, d)
2      SN_d ← SNL[d];                          //get weighted sampling number for block d
3      SB_V ← randomSampling (SBL[d], SN_d);   //sampling SN_d seed block values
4      foreach v in SB_V do
5          p[d] ← v;                           //insert v to the position d of p
6          if d == SBL.size then
7              insert (PL, p);                 //a full path generated
8          else
9              getPathByDF (SBL, SNL, p, d + 1); //recursively process to depth d+1
10     return PL
```

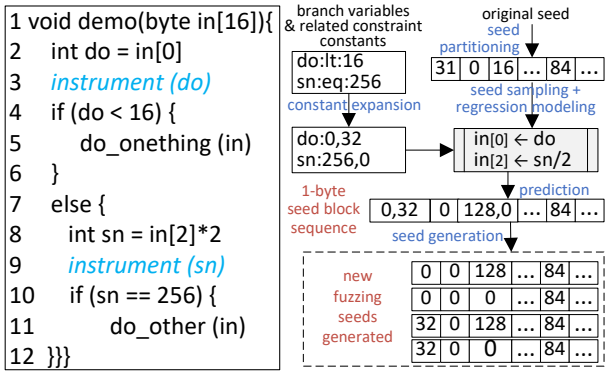

Figure 5: An example illustrating **Phase** 2.

To illustrate **Phase** 2 as a whole, consider the example of Figure 5. Two branch variables (i.e., do and sn) are instrumented at their definition sites. After seed partitioning (for block size=1) and sampling, the regression modeling learns models each for one SBP, of which two are shown here: $in[0] \leftarrow do$ and $in[2] \leftarrow sn/2$—each model approximates a function mapping a branch variable to a seed block in an SBP. Then, the constraint constants for the two branch variables are expanded according to the operator type (e.g., less than (lt), equal (eq)). In particular, the constant 16 for the branch variable do is expanded to two values of do: 0 which satisfies the constraint hence makes the branch predicate true and 32 in the other direction. Similarly, sn also obtains two values (i.e., 256, 0). Next, the learned functions take the values of do to predict values of seed block $in[0]$ while using the values of sn to predict values of the seed block $in[2]$. Finally, applying Algorithm 5 on the block sequence leads to four new seeds.

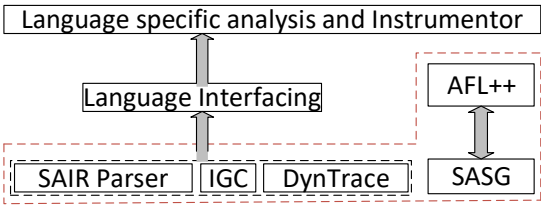

Figure 6: An overview of PolyFuzz's implementation.

3.4 Core Fuzzing (Phase 3)

In this phase, when the fuzzing is activated (i.e., the **Fuzzing mode** of **Phase** 2 is alive), PolyFuzz runs the conventional fuzzing (including seed selection, mutation, and bug reporting) algorithm. With all the basic block information of different language units mapped to the same shared memory byte-map, PolyFuzz calculates the block and path coverage without knowing about the languages used in the program under test, making the core fuzzing language-agnostic.

To coordinate between **Phase** 2 and **Phase** 3, PolyFuzz includes a submodule in the core fuzzing module to control the fuzzing to start, to either become/stay idle or active, or to load new seeds for the next fuzzing iteration.

4 Implementations and Limitations

We have implemented PolyFuzz to support programs developed in one or more of three popular languages: Python, Java, and C. Figure 6 shows the key components of PolyFuzz. It has a common C component for static analysis and instrumentation, including three basic libraries: SAIR parser, instrumentation guidance computation (IGC), and DynTrace. The language-specific analysis layer can use these libraries through the wrapper in the language interfacing layer. It also has a component for sensitivity analysis and seed generation (SASG) which realizes **Phase** 2. For the core fuzzing (**Phase** 3), PolyFuzz uses AFL++ [16] as its fuzzing core. The entire implementation includes 12KLoC (0.6 KLoCfor Java, 1.2 KLoCfor Python, and 0.3KLoC for C). Further details on the implementation can be found in Appendix A.

4.1 Supporting Other Languages

The lighter-weight, *fuzzing-specific* custom IR (i.e., SAIR), which only requires minimal language-specific analysis and instrumentation, makes the rest of PolyFuzz language-independent, hence allowing for the meritorious extensibility of PolyFuzz in terms of supporting other languages and language combinations. In particular, given a new language to support, only the language-specific analysis and instrumentor as shown in Figure 6 need to be added, as elaborated below.

Language-specific analysis. The goal of this analysis is SAIR translation for the new language. Based on its definition (§3.2), the translator summarizes the type of each variable (`Integer` vs. `Other`) on each statement while particularly identifying branching statements and branch variables. Thus, programs of imperative languages (e.g., Ruby and Javascript) can be readily translated to SAIR hence supported by PolyFuzz with the support of a respective parser (e.g., tree-sitter-ruby [67], tree-sitter-javascript [66]), while declarative languages (e.g., SQL, HTML) may not. As a reference, for the implementation of PolyFuzz we use around 150 lines of code to translate Java to SAIR.

Language-specific instrumentor. With the instrumentation guidance computed by the (language-independent) IGC module based on the SAIR translation, the language-specific instrumentor for the new language probes for basic blocks and branch variables. As a reference, in PolyFuzz the instrumentor for Java is implemented in around 80 lines of code.

4.2 Limitations

To ensure a pure run-time environment for each execution of the fuzzing target, PolyFuzz always works in non-persistent mode [16], in which PolyFuzz forks a new process for each fuzz execution. This kind of implementation makes PolyFuzz easy to use and stable, since users need not consider whether the target is stateless and PolyFuzz would not stop when a bug is triggered during fuzzing. However, this implementation also has a drawback: the process of forking can affect the target execution hence fuzzing efficiency.

In addition, PolyFuzz is currently implemented under an assumption that the fuzzing target runs in a single process; otherwise, the coverage collected may be misleading since interprocess information flow [7] will be missing in the coverage feedback. According, cross-process bugs [18] may be difficult for PolyFuzz to trigger. This limitation would potentially render PolyFuzz insufficient for fuzzing multilingual code with an entry language unit written in C, because handling such code would often involve *multi-process* fuzzing (e.g., C unit would invoke other language units via separate processes). Meanwhile, during our experiments with PolyFuzz, we tried hard but failed to find much of multi-language systems with C entries—we did not find any from GitHub when collecting evaluation benchmarks (*S*5.1). Such a rare presence suggests a relatively minor impact of this limitation of PolyFuzz on systems using C as a main language.

We offer (in §6) further insights into several decisions and tradeoffs in our design and implementation of PolyFuzz.

5 Evaluation

We evaluated PolyFuzz through answering the following three research questions:

RQ1 *How effective is* PolyFuzz *on real-world multilingual programs vs state-of-the-art single-language fuzzers?* (§5.2)
RQ2 *How effective is* PolyFuzz *on single-language programs vs state-of-the-art single-language fuzzers?* (§5.3)
RQ3 *How important is the sensitivity analysis based seed generation (SASG) in* PolyFuzz*?* (§5.4)

Table 1: Profiles of 15 real-world multi-language systems used as our subjects (Size in KLOC, BV: branch variable, BV-IntConst: branch variable with constant integer constraints)

Benchmark	Size	Languages		#BV	#BV-IntConst
Libsmbios [12]	8.3	Python:30.4%	C:64.2%	6866	3269 (47.6%)
Tink [21]	257.7	Python:7.2%	C++:33.5%	66282	27962 (42.2%)
Pillow [56]	75.8	Python:60.0%	C:38.6%	15628	9090 (58.2%)
Ultrajson [69]	5.1	Python:34.3%	C:64.8%	1361	903 (66.1%)
Aubio [4]	42.9	Python:25.4%	C:73.3%	3445	2232 (64.8%)
Bottleneck [59]	16.9	Python:49.5%	C:48.6%	3384	1814 (53.6%)
Pycurl [58]	14.6	Python:54.8%	C:40.7%	433	264 (61.1%)
Simplejson [63]	6.2	Python:61.4%	C:38.6%	858	544 (63.4%)
Msgpack [48]	15.1	Python:48.7%	C:50.1%	2322	1056 (45.5%)
Pycryptodome [36]	65.5	Python:43.5%	C:56.1%	2842	1595 (56.1%)
Jep [51]	18.9	Java:25.4%	C:56.6%	2856	1454 (50.9%)
Jansi [19]	5.2	Java:66.6%	C:22.7%	386	121 (31.3%)
Jna [32]	129.4	Java:76.9%	C:16.1%	3017	941 (31.2%)
One-nio [52]	29.1	Java:86.0%	C:14.0%	4371	1132 (25.9%)
Zstd-jni [44]	47.9	Java:6.8%	C:88.7%	47803	20384 (42.6%)

5.1 Experiment Setup

Experiment Environment. All experiments were conducted on a machine running 64-bit Ubuntu 18.04 with a 32-core CPU (AMD Ryzen Threadripper 3970X) and 256 GB memory. We ran each fuzzer against each target application with identical configurations on one CPU core for 24 hours. All experiments were repeated 5 times.

Baseline Fuzzers for Comparison. PolyFuzz was compared to three state-of-the-art single-language fuzzers used in Google OSS-Fuzz framework [23], i.e., Honggfuzz [65] for C, Jazzer [11] for Java, and Atheris [22] for Python. Since Jazzer and Atheris are not originally designed for fuzzing multi-language systems, coverage of C (e.g., native) code in such systems is not automatically probed for and measured. Yet comparing PolyFuzz with the extended versions of these baselines (i.e., with C code coverage probed for and measured) can help better evaluate the design of PolyFuzz. Thus, we also developed and compared PolyFuzz to the extended versions of Jazzer and Atheris with any covered C code measured, referred to as Jazzer-C-ext and Atheris-C-ext, respectively. Moreover, to evaluate the effectiveness of SASG, we compared PolyFuzz to a downgraded version of PolyFuzz with SASG disabled (noted as PolyFuzz-NSA) on multilingual benchmarks and to AFL++ on C benchmarks.

Benchmarks and Initial Input Seeds. We evaluated PolyFuzz against 15 real-world multilingual systems, including 10 Python-C and 5 Java-C programs. Table 1 summarizes these systems as our subjects (1st column), including the code size (2nd column), language distribution (3rd column), the number of branch variables (4th column), and the number and percentage of integer-constant-constrained branch variables (last column). All of these 15 systems were downloaded from GitHub with high popularity and frequent updates.

Table 2: The 15 single-language systems randomly selected from Oss-Fuzz (Size in KLOC, BV: branch variable, BV-IntConst: branch variable with constant integer constraints))

Benchmark	Size	Languages	#BV	#BV-IntConst
Bleach [47]	14.4	Python	1035	119 (11.5%)
Sqlalchemy [64]	391.9	Python	30637	2187 (7.1%)
Urllib3 [70]	18.5	Python	1948	121 (6.2%)
Pyyaml [74]	24.3	Python	2196	107 (4.9%)
Pygments [60]	96.6	Python	4993	381 (7.6%)
Json-sanitizer [53]	2.3	Java	326	237 (72.7%)
Commons-compress [2]	73.7	Java	8563	5771 (67.4%)
Zxing [80]	47.1	Java	4453	3059 (68.7%)
Jsoup [30]	25.3	Java	2109	1101 (52.2%)
Javaparser [29]	183.9	Java	7743	4683 (60.5%)
E2fsprogs [68]	118.4	C	19439	13279 (68.3%)
Bind9 [28]	275.4	C	56428	33538 (58.8%)
Civetweb [10]	521.7	C	6615	4080 (61.7%)
Cyclonedds [14]	225.9	C	22551	14286 (63.3%)
Igraph [27]	212.1	C	63013	35043 (55.6%)

Moreover, to evaluate PolyFuzz's performance on single-language projects, we randomly selected 5 real-world benchmarks from Google Oss-Fuzz for C, Python, and Java, respectively, as shown in Table 2 in a similar format to Table 1.

Regarding fuzzing drivers and initial seeds, we developed new drivers for PolyFuzz on all the 15 multilingual systems, while for Atheris on the 10 Python-C programs and for Jazzer on the 5 Java-C programs. We did use the same drivers (in terms of targeted APIs/code) across all of the fuzzers considered in order to ensure fair comparisons; we had to *adapt* the drivers for different fuzzers given PolyFuzz's different test-input interface from other fuzzers. As C units in these projects are all internal libraries, we did not develop drivers for C APIs for reasons discussed in §2. For all the 15 single-language projects, we reused the drivers in Oss-Fuzz for all the single-language fuzzers, and developed new drivers for PolyFuzz. To ensure the fairness in evaluation, we ran all fuzzers on the same benchmark with the same initial inputs.

Performance Metrics. We considered two common fuzzing metrics: #basic blocks covered and #bugs triggered. The three baseline single-language fuzzers all support using basic block coverage as feedback, hence we used the #basic blocks as a main performance evaluation indicator. As PolyFuzz uses AFL++ [16] as the core fuzzing agent, we also reported the #paths identified by AFL++'s algorithm as reference. For the comparison between PolyFuzz and PolyFuzz-NSA, we used the #paths found as the third metric. The coverage results were averaged based on 5 repetitions of 24-hour running.

Another important metric is the #bugs detected. Since the number of unique crashes reported by different fuzzers may be inaccurate, we manually validated all reported issues. Specifically, we developed a PoC to reproduce each issue with the

crash-triggering inputs. If the crash can be reproduced, then we consider a crash as a new bug only when its call stack differs from all other bugs that have been confirmed.

5.2 RQ1: Effectiveness of PolyFuzz on Multilingual Programs

Table 3 shows the results of PolyFuzz versus Atheris and Atheris-C-ext on the 10 Python-C benchmarks. For a fair comparison, we report both the total #basic blocks covered (#*Block*) and #basic blocks covered in the Python unit (#*PythonBlk*) by PolyFuzz. With Atheris, only Python code coverage (#*PythonBlk*) is measured and used as feedback, while with Atheris-C-ext the coverage (#*Block*) additionally includes any C code covered. Similarly, Table 4 presents the comparison results among PolyFuzz, Jazzer, and Jazzer-C-ext on the 5 Java-C benchmarks, except for #*PythonBlk* being changed to #*JavaBlk* to indicate #basic blocks covered in the Java unit. Since none of the multilingual benchmarks have its entry in their C units, we could not run PolyFuzz and Honggfuzz for comparison for this RQ.

Coverage. PolyFuzz covers 36.7% more basic blocks in the whole system than Atheris-C-ext, and 52.3% more basic blocks in the Python units than Atheris, as shown in Table 3. Compared to Jazzer, these two numbers are 25.3% and 29.1%, respectively, as shown in Table 4. These results reveal that PolyFuzz substantially improves the code coverage both in the whole system and in the comparable language units.

Among the multilingual benchmarks, the C code accounts for 38.6% (Simplejson) to 73.3% (Aubio) of the Python-C program sizes, and 14.0% (One-nio) to 88.7% (Zstd-jni) of the Java-C programs sizes, as per Table 1. However, both Atheris and Jazzer treat the C units as black boxes. Therefore, they are not sensitive to the coverage changes in these units during fuzzing. The incomplete coverage guidance makes it difficult for the two single-language fuzzers to evolve and leads them to soon get stuck, as we observed in the experiments. With C code coverage measured/used, both Atheris-C-ext and Jazzer-C-ext intuitively have more code blocks covered in total.

By contrast, PolyFuzz utilizes the whole-system coverage as feedback; hence it can identify more favored seeds that may be ignored in the single-language fuzzers when the coverage change occurs in C units. Furthermore, by mutating these seeds, PolyFuzz can evolve more efficiently and promotes the coverage of the whole system. Hence, it achieved much higher coverage than Atheris and Jazzer. Moreover, by learning the regression models to capture semantic relationships between branch variables and seed blocks, PolyFuzz is capable of generating more effective seeds to exercise the related branch predicates in both directions, further facilitating the exploration of more branches hence that of more program paths. And because of that, PolyFuzz can cover more blocks than both Atheris-C-ext and Jazzer-C-ext also.

Table 3: Performance comparison among PolyFuzz, Atheris, and Atheris-C-ext against the Python-C benchmarks.

Benchmark	PolyFuzz				Atheris		Atheris-C-ext	
	#Block	#PythonBlk	#Path	#Bug	#PythonBlk	#Bug	#Block	#Bug
Libsmbios	198	51	35	1	24	0	149	0
Tink	2139	97	755	0	33	0	1891	0
Pillow	1363	1034	233	1	706	1	915	1
Ultrajson	377	126	151	1	39	0	238	0
Aubio	453	187	91	1	126	0	160	0
Bottleneck	1359	25	634	7	14	0	886	2
Pycurl	239	38	19	0	26	0	205	0
Simplejson	374	97	86	0	82	0	197	0
Msgpack	245	48	78	0	43	0	223	0
Pycryptodome	572	243	64	0	185	0	493	0
Total	7319	1946	2147	11	1278	1	5357	3
Improve	-				52.3% ↑	10 ↑	36.7% ↑	8 ↑

Bug triggering. As shown in Table 3 and Table 4, Atheris triggered 1 bug in project Pillow and Atheris-C-ext further triggered 2 in Bottleneck while Jazzer and Jazzer-C-ext failed to find any bugs. Remarkably, PolyFuzz succeeded in triggering 12 bugs in 6 projects, including 11 in Python-C and 1 in Java-C programs. For all the 12 bugs, we have manually confirmed and developed proof-of-the-concept (PoC) for reproduction. The whole-system coverage awareness in PolyFuzz not only promotes the evolution of fuzzing process to gain high code coverage but also increases the possibility of discovering bugs in real-world multi-language projects.

Table 4: Performance comparison among PolyFuzz, Jazzer, and Jazzer-C-ext against the Java-C benchmarks.

Benchmark	PolyFuzz				Jazzer		Jazzer-C-ext	
	#Block	#JavaBlk	#Path	#Bug	#JavaBlk	#Bug	#Block	#Bug
Jep	418	145	59	0	90	0	354	0
Jansi	332	309	244	1	242	0	261	0
Jna	711	476	189	0	362	0	579	0
One-nio	364	316	131	0	289	0	312	0
Zstd-jni	151	84	21	0	47	0	71	0
Total	1976	1330	644	1	1030	0	1577	0
Improve	-				29.1% ↑	1 ↑	25.3% ↑	1 ↑

PolyFuzz achieved 25.3%–52.3% higher block coverage and discovered 1–10 more bugs than state-of-the-art single-language fuzzers against real-world multi-language systems, for all the languages the current PolyFuzz implementation supports (i.e., Java, C, Python).

5.3 RQ2: Effectiveness of PolyFuzz on Single-Language Programs

Next, we compare PolyFuzz with Atheris, Jazzer and Honggfuzz against the 15 real-world single-language benchmarks.

Coverage. As shown in Tables 5-7, PolyFuzz covered 20.1%, 11.0% and 10.1% more basic blocks than Atheris, Jazzer and Honggfuzz on these single-language benchmarks, respectively. Unlike against the multi-language benchmarks,

Table 5: Performance comparison between PolyFuzz and Atheris on the Python benchmarks.

Benchmark	PolyFuzz			Atheris	
	#Block	#Path	#Bug	#Block	#Bug
Pyyaml	853	703	1	826	1
Bleach	1023	353	0	796	0
Sqlalchemy	1096	18	0	1047	0
Pygments	1276	229	0	799	0
Urllib3	534	71	0	496	0
Total	4782	1474	1	3964	1
Improve	-			20.1% ↑	0 –

Table 6: Performance comparison between PolyFuzz and Jazzer on the Java benchmarks.

Benchmark	PolyFuzz			Jazzer	
	#Block	#Path	#Bug	#Block	#Bug
Zxing	4604	1923	0	4575	0
Jsoup	3408	1082	0	3261	0
Javaparser	4729	377	1	3821	1
Commons-compress	339	453	0	296	0
Json-sanitizer	595	309	0	547	0
Total	13675	4144	1	12319	1
Improve	-			11.0% ↑	0 –

all fuzzers can use the complete, whole-system coverage as feedback here. Nevertheless, PolyFuzz still exhibited better performance than all the 3 baseline fuzzers. As shown in Table 2, all of these benchmarks have a notable portion (ranging from 4.9% in Pyyaml to 72.7% in Json-sanitizer) of branch variables with constant integer constraints (5th column). PolyFuzz's SASG module enables effective seed generation from these constant branch constraints. Further, with the generated seeds as inputs, PolyFuzz can cover more blocks with fewer random mutations. Overall, PolyFuzz was able to discover more favored seeds by further mutating these seeds, which are generated with seed blocks that have a strong association with branch variables.

Bug triggering. PolyFuzz successfully triggered 2 new bugs, including 1 `Recursion error` in the Python benchmark Pyyaml and 1 `JVM hung` in the Java benchmark Javaparser, and did not trigger any bugs in the C benchmarks. For the bug in Pyyaml, Atheris also reported a similar issue with a different seed input. We developed PoC for and reproduced the bug with both seeds. Through manually validating the call stacks, we confirmed PolyFuzz and Atheris triggered the same bug. Similarly, the bug discovered by Jazzer was confirmed as the same as triggered by PolyFuzz. Although PolyFuzz did not show an overwhelming advantage when compared with these single-language fuzzers in terms of bug triggering in our experiments, PolyFuzz still has more potential for bug discovery due to the higher code coverage.

Table 7: Performance comparison between PolyFuzz and Honggfuzz on the C benchmarks.

Benchmark	PolyFuzz			Honggfuzz	
	#Block	#Path	#Bug	#Block	#Bug
E2fsprogs	1173	302	0	1049	0
Bind9	4154	2982	0	3955	0
Civetweb	232	157	0	195	0
Cyclonedds	1091	592	0	1003	0
Igraph	431	454	0	228	0
Total	7081	4457	0	6430	0
Improve	-			10.1% ↑	0 –

Table 8: Performance comparison between PolyFuzz and PolyFuzz-NSA on 15 multilingual programs.

Benchmark	PolyFuzz			PolyFuzz-NSA		
	#Block	#Path	#Bug	#Block	#Path	#Bug
Libsmbios	198	35	1	174	30	1
Tink	2139	755	0	1771	627	0
Pillow	1363	233	1	1043	147	1
Ultrajson	377	151	1	318	120	0
Aubio	453	92	1	349	85	0
Bottleneck	1359	634	7	1321	516	2
Pycurl	239	19	0	198	18	0
Simplejson	374	86	0	239	54	0
Msgpack	245	78	0	201	67	0
Pycryptodome	572	64	0	469	50	0
Jep	418	59	0	364	51	0
Jansi	332	244	1	313	211	0
Jna	711	189	0	671	181	0
One-nio	364	131	0	343	119	0
Zstd-jni	151	21	0	145	19	0
Total	9295	2791	12	7853	2292	4
Improve	-			17.4% ↑	21.8% ↑	8 ↑

Despite not aiming at single-language fuzzing, PolyFuzz *still covered 10.1-20.1% more basic blocks than, and triggered the same #bugs as, the three studied state-of-the-art single-language fuzzers.*

5.4 RQ3: Importance of SASG in PolyFuzz

Both PolyFuzz and PolyFuzz-NSA support cross-language fuzzing, albeit PolyFuzz-NSA only benefits from holistic coverage feedback. So we compared the two fuzzers on the 15 real-world multilingual programs in terms of three performance metrics: #basic blocks (#*Block*), #paths (#*Path*), and #bugs triggered (#*Bug*). The results are shown in Table 8. To assess the merits of SASG for single-language fuzzing, we also compared PolyFuzz and AFL++ on the five C benchmarks, with results summarized in Table 9.

Coverage. In terms of both basic block and path coverage, PolyFuzz has a clear advantage over PolyFuzz-NSA. For

Table 9: Performance comparison between PolyFuzz and AFL++ on the C benchmarks.

Benchmark	PolyFuzz			AFL++		
	#Block	#Path	#Bug	#Block	#Path	#Bug
E2fsprogs	1173	302	0	1066	217	0
Bind9	4154	2982	0	3996	2813	0
Civetweb	232	157	0	198	145	0
Cyclonedds	1091	592	0	1016	531	0
Igraph	431	454	0	308	322	0
Total	7081	4457	0	6584	4028	0
Improve	-			7.6% ↑	11.4% ↑	0 –

instance, against Pillow, PolyFuzz exercised 320 (30.7% ↑) more basic blocks and 86 (58.5% ↑) more paths than PolyFuzz-NSA. Overall, PolyFuzz covered 17.4% more basic blocks and 21.8% more paths. These results indicate that the novel sensitivity analysis and seed generation techniques in PolyFuzz contributed significantly to its overall cost-effectiveness and superiority over the baselines. Nonetheless, PolyFuzz-NSA was still capable of covering more basic blocks than the single-language fuzzers, due to the holistic coverage awareness. Specifically, PolyFuzz-NSA covered 30.5% (versus 52.3% by PolyFuzz) more Python blocks than Atheris and 19.5% (versus 29.1% by PolyFuzz) more Java blocks than Jazzer. When compared to AFL++ on the C benchmarks, PolyFuzz covered 7.6% more basic blocks and 11.4% more paths under the same coverage feedback mechanism. As PolyFuzz uses AFL++ as the core fuzzer, this comparison indicates the *general* merits of SASG in PolyFuzz beyond multilingual fuzzing (i.e., the merits apply to single-language fuzzing as well). Thus, *our SASG could be incorporated into existing single-language fuzzers to significantly improve their performance as well.*

Bug triggering. PolyFuzz-NSA only triggered 4 of the 12 bugs detected by PolyFuzz. Thus, the SASG brought a strong improvement in bug-finding power to PolyFuzz. On the other hand, PolyFuzz-NSA still triggered more bugs than single-language fuzzers, per Table 8 vs Tables 3 and 4. Between AFL++ and PolyFuzz, both failed to detect any bugs within the given time, as did Honggfuzz—after all, code coverage does not always lead to bug discovery.

A case study on coverage growth. Figure 7 depicts the trend of growth in #basic blocks and #paths covered over the 5 runs of 24-hour experiments on Pillow. PolyFuzz performed a little weaker initially, since SASG can quickly run into the Learning mode when observing new branch variables covered, at the moment, any branch variables are new to SASG. It will take a while for SASG to do seed partitioning and sampling for these branch variables, during which the core fuzzing must stay idle. Once the first regression learning pass is over, the number of basic blocks and paths can quickly grow based on the learned seeds. Moreover, the learning keeps going during the fuzzing campaign, promoting the coverage to keep its sustained growth. By contrast, PolyFuzz-NSA ran into a stalemate after running for 12 hours.

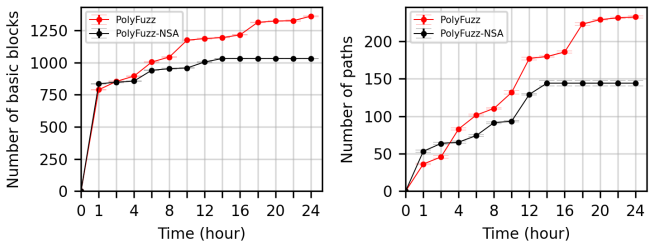

Figure 7: The growing trend of the average number of basic blocks and paths covered during the 5 24-hour runs on Pillow.

Table 10: Vulnerabilities detected by PolyFuzz.

Benchmark	#Bug	Status	PoC	Symptom	#CVE
Libsmbios	1	pending	✓	segment fault	0
Pillow	1	fixed	✓	out of memory	1
Ultrajson	1	fixed	✓	segment fault	1
Aubio	1	pending	✓	memory leak	0
Bottleneck	7	pending	✓	segment fault	1
Jansi	1	pending	✓	out of memory	1
Pyyaml	1	pending	✓	recursion error	0
Javaparser	1	confirmed	✓	JVM hung	1
Total	14	-			5

SASG contributed significantly to PolyFuzz's performance in terms of both coverage and bug triggering; it also has general merits for both multilingual and single-language greybox fuzzing.

5.5 Regarding the Vulnerabilities Discovered

Table 10 summarizes the total of 14 new vulnerabilities discovered by PolyFuzz during our evaluation. We have developed PoCs for these vulnerabilities to ensure reproducibility and contacted the respective developers. By the time of the paper submission, all of these have been reported to the system vendors, and two of them have been confirmed and fixed. The details on each of these 14 vulnerabilities are documented in **NewVulnerabilities.pdf** within our artifact package.

As an example of these new vulnerabilities, Figure 8 shows a NULL pointer dereference in Ultrajson. In the Python unit, the input is read into the variable `data` and then passed to the `ujson` API `ujson.dump`. Then, the value of `data` flows forward into the C function `SortedDict_iterNext` wrapped in the JSON object `obj`. After the variable `key` is decoded from `obj` and passes the unicode check (line 8), function `PyUnicode_AsEncodeString` tries to encode it. However, this function can return NULL with a specific input and further causes NULL pointer access at line 12. This can be exploited to enable Denial of Service (DoS) attacks constantly crashing the program with carefully crafted inputs.

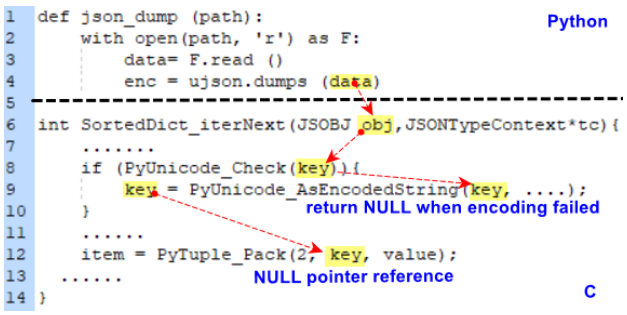

Figure 8: New vulnerability case: NULL pointer dereference.

6 Discussion

The first, intuitive contributing element in the design of PolyFuzz is *holistic coverage measurement and feedback*. In this regard, one may wonder the multilingual fuzzing problem dealt with by PolyFuzz could have been addressed by combining single-language fuzzers for all the languages used by the multi-language system in question. However, applying the combination of single-language fuzzers would suffer feasibility/reproducibility challenges (§2.2) and hence be insufficient for discovering *cross-language* bugs. Also, for non-entry language units, developing good drivers can be as hard as library fuzzing while requiring cross-language calling relationship analysis.

As the other major contributor to its superior capabilities, PolyFuzz utilizes sensitivity analysis to model the relationships between inputs and integer branch variables; then with the expanded constant branch constraint as inputs, it generates new seeds to hit or reverse respective branches effectively. The newly covered code, in turn, can trigger a new round of seed learning, empowering continuous coverage growth during fuzzing. In our evaluation, PolyFuzz demonstrated exemplary performance and effectiveness in fuzzing both multi- and single-language programs.

Meanwhile, several factors may limit PolyFuzz in practice. First, there are still significant proportions of non-integer branch variables in real-world programs (see Tables 1 and 2), for which PolyFuzz currently would not work as effectively. Second, The correlation between inputs and branch variables may be too complicated to model by a regression model of any types, or even does not exist. In those cases, SASG may not help. Third, in the seed partitioning and sampling phase, the execution paths may vary during the random mutation, affecting the branch variable coverage, which may cause failures to collect enough data for training the regression models. Finally, long seeds can limit the fuzzing performance since it can make PolyFuzz stay in seed sampling for too long, during which the core fuzzing is disabled.

PolyFuzz mitigates part of the above limitations in its implementation. Specifically, when new paths are covered during the sensitivity analysis, PolyFuzz doubles the sampling size (i.e., $\mathbb{N}$ taken by Algorithm 3) trying to harvest enough branch variable values. Of course, this drills down to a tradeoff between core fuzzing time and model learning effectiveness—since the fuzzing is suspended during the sampling. To mitigate the effects of long seeds, PolyFuzz limits the length for sampling to 2K bytes, as our empirical study found that over 90% of the successful seed learning happened in the first 2k bytes of the inputs over the tested benchmarks.

Dealing with non-integer branch variables can be much more complicated, since data representation is different over various languages, e.g., a Python object (e.g., a `list`) or a Java object (e.g., a `hashtable`). Two key challenges hinder the SASG on these variables: (1) *how do we unify the data representation among different languages?* Differences in data representation can prevent effective instrumentation of programs. (2) *what kind of data storage is efficient?* The data size of a non-integer variable can be huge (e.g., a Python list).

For the cases in which no correlation between branch variables and seed blocks can be learned, we sampled some branch variables for a manual validation. We found that such variables fall mainly into two categories: (1) the return value of functions, where only specific values are taken (e.g., 0 or -1); (2) the value fetched from configurations (e.g., read from environment variables or configuration files). How fuzzing may automatically incorporate the change of these configurations is a topic worthy of study; Such solutions could significantly improve code coverage and bug discovery.

One common design concern in cross-language analyses is language independence hence *extensibility* [40] in terms of the ability and ease to support other languages, language combinations, and corresponding interfacing mechanisms [38]. Accordingly, in multilingual fuzzing, there is a tradeoff between the extensibility and the ability/efficiency for exercising bugs right at cross-language interfaces. For instance, assuming and utilizing knowledge about specific interface layers and any additional checks (e.g., predicates guarding against a native/foreign function call) therein may help improve fuzzing efficiency, but it would also compromise the fuzzer's extensibility. For PolyFuzz, our design favors extensibility hence assumes no such knowledge. As a result, our core fuzzing process is language-agnostic and treats those checks transparently as any other such language constructs within individual language units. On the other hand, analysis of specific language interface layers may be implemented on top of PolyFuzz.

7 Related Work

Single-language fuzzing. Most existing fuzzers target C/C++ [46]. Particularly for greybox fuzzing, MOpt [45] optimizes mutation operator selection using a particle swarm optimization algorithm; GreyOne [20] and REDQUEEN [3] use dynamic taint analysis to guide where and how to mutate; ProFuzzer [76] probes for critical bytes in the input to adapt the mutation strategy. PATA [41] also identifies critical input

bytes, but for approximating the relationships between input and constraint variables hence enabling path-aware mutation.

In addition to *seed mutation*, other fuzzers focus more on efficient *seed generation* [8,71,77] and *seed selection/prioritization* [5,54,78]. Also, beyond program analysis, alternative approaches have been explored to improve these key algorithmic components of fuzzing (e.g., using reinforcement learning for seed scheduling [72], transforming code to remove input sanity checks that the fuzzer gets stuck with [55]). In summary, PolyFuzz differs from prior seed-generation works in four aspects: (1) peer prior works treat input as byte-stream, versus seed-block stream in PolyFuzz; (2) prior works gain more seeds *after* mutation, versus PolyFuzz predicting seed-block values via regression modeling followed by assembling them to form new seeds; (3) our seed generation is based on sensitivity analysis with regression modeling at its core, different from taint analysis guided seed generation/mutation; and (4) our regression model is selected adaptively on the fly, not necessarily linear (see Algorithm 4) as in Eclipser [9].

There have been only a very few fuzzers for other languages in the literature. Both based on libFuzzer [43], Jazzer [11] works for Java and Atheris [22] for Python. Also working for Java applications, Kelinci [33] feeds AFL [49] with instrumented Java bytecode and DiffFuzz [50] directly adopts AFL for differential testing. Rust-fuzz [62] and Rulf [31] adapt AFL/AFL++ for fuzzing Rust application/libraries.

In comparison, while (naturally) also supporting fuzzing single-language programs, PolyFuzz uniquely addresses fuzzing multi-language software in a *holistic* manner. In addition to its coverage monitoring and feedback mechanism across languages, it also differs from peer techniques in explicitly modeling the semantic relationship between (different segments of) inputs and branch predicates. Eclipser [9] also models such relationships, but it only addresses those that are linear or monotonic, while our regression model is selected adaptively and optimally on the fly, from among linear, rbf, and polynomial models—not necessarily linear or monotonic. Moreover, when generating new inputs, Eclipser only considers one input field, ignoring the effects of various combinations of different fields, as opposed to PolyFuzz modeling the effects of seed input blocks of varying sizes. Finally, the seed generation in PolyFuzz is based on sensitivity analysis, of which seed partitioning and sampling, constant expansion, and seed-block assembling are all integral parts in addition to regression modeling, unlike Eclipser generating new inputs by solving linear equations/inequalities and binary search.

Multi-language testing. Amleto [15] is an embedded software testing platform that supports both VHDL and SystemC by translating both to a custom internal intermediate representation (IR). Similarly, Gillian [17] provides a framework for multi-language symbolic execution also based on an IR (named GIL). It was implemented for JavaScript and C, which are converted to the GIL for symbolic testing. The language-specific analysis and memory model can be customized by users for a particular target language. The mutation testing tool in [25] supports mutant generation for different languages through regular-expression-based source code transformations. None of these tools actually target programs each consisting of code units in multiple languages at the same time, hence clearly different from PolyFuzz supporting the testing of *multilingual* code.

FANS [42] offers the capability of fuzzing Android native system services (mainly written in C++) which may invoke Java code. While it indirectly triggered a number of Java exceptions, FANS itself works as a single-language (C++) fuzzer without dealing with cross-language code. Favocado [13] aims to fuzz JavaScript engines particularly focusing on their binding layer, which translates data between JavaScript and low-level languages like C/C++. Yet the binding code itself is still written in C/C++. Thus, like FANS, Favocado is a single-language fuzzer, rather than fuzzing JavaScript code interfaced with C/C++.

We are not aware of prior work explicitly addressing holistic fuzzing of multilingual code as PolyFuzz does.

Cross-language security analysis. NDroid [73] provides a QEMU-based dynamic taint analysis (DTA) for JNI code (i.e., in Java and C) in Android apps, which enabled discovery of cross-language information leakage. More recently, Li et al. developed PolyCruise [40], a purely application-level dynamic information flow analysis (DIFA) which has helped detect a number of vulnerabilities at language interfacing between Python and C/C++ units. Despite their (demonstrated) potential for finding security vulnerabilities, these cross-language analyses rely on existing run-time inputs that trigger the executions underlying the dynamic analysis. Also, given their design, it would take considerable effort to extend these tools to support other language combinations.

In contrast, PolyFuzz lifts such limitations by generating the inputs that are needed to trigger vulnerabilities in multi-language software, while offering better extensibility. On the other hand, cross-language DTA/DIFA as presented in the above prior works may be leveraged to guide holistic multi-language fuzzing like PolyFuzz.

Program intermediate representation (IR). Traditional IRs (e.g., LLVM-IR [35], Soot/Jimple [34]) serve whole-program translation fully covering the original code semantics, which is too heavyweight and even impractical for various languages when targeting a *unified* IR. The custom IR in PolyCruise [40] is a symbolic representation serving *data-flow* analysis [37], which is also unnecessarily heavyweight for greybox fuzzing. In contrast, our new custom IR is a much simpler/lighter-weight, *fuzzing-specific* IR just capturing fuzzing-relevant information such as control-flow/branching and value types—which are not considered in the PolyCruise IR.

On a related note, this new custom IR in PolyFuzz is motivated and justified by its ability to maximally support different language combinations hence offering the extensibility to support other languages [75]. PolyFuzz employs a minimal

language-specific analysis for holistic coverage measurement and harvesting only the variable values necessary for learning the regression model. This is enabled by this new custom IR that unifies run-time value probing across heterogeneous languages, which makes the rest (and most) of PolyFuzz language-agnostic, hence addressing the *Challenge-2* (§1).

8 Conclusion

We presented PolyFuzz, a novel framework for holistic grey-box fuzzing of multi-language software. PolyFuzz measures whole-system block and path coverage in a language-agnostic as enabled by a custom intermediate representation particularly designed for fuzzing. Beyond the holistic coverage feedback, it also generates new seeds effectively via regression modeling the semantic relationships between seeds and branch variables. Our results reveal significant merits of PolyFuzz over state-of-the-art single-language fuzzers.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their effective guidance and constructive comments. This work was supported in part by National Science Foundation (NSF) under Grant CCF-2146233 and in part by Office of Naval Research (ONR) under Grant N000142212111.

References

[1] SourceForge: The Complete Open-Source and Business Software Platform. https://sourceforge.net, 2020.

[2] Apache. Apache Commons Compress. https://github.com/apache/commons-compress, 2022.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[4] aubio. A library to label music and sounds. https://github.com/aubio/aubio.git, 2019.

[5] 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.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[7] Haipeng Cai and Xiaoqin Fu. D^2ABS: A framework for dynamic dependence analysis of distributed programs. *IEEE Transactions on Software Engineering*, 2021.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[9] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE, 2019.

[10] civetweb. An embeddable web server with optional CGI, SSL and Lua support. https://github.com/civetweb/civetweb, 2022.

[11] Code-Intelligence. Coverage-guided, in-process fuzzing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer, 2022.

[12] dell. A library to interface with the SMBIOS tables. https://github.com/dell/libsmbios, 2007.

[13] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[14] Eclipse Cyclone DDS. An open-source implementation of the OMG DDS specification. https://github.com/eclipse-cyclonedds/cyclonedds, 2022.

[15] Alessandro Fin, Franco Fummi, and Graziano Pravadelli. Amleto: A multi-language environment for functional test generation. In *Proceedings International Test Conference*, pages 821–829. IEEE, 2001.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[17] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 927–942, 2020.

[18] Xiaoqin Fu and Haipeng Cai. FlowDist: Multi-staged refinement-based dynamic information flow analysis for distributed software systems. In *30th USENIX Security Symposium (USENIX Security)*, pages 2093–2110, 2021.

[19] fusesource. A java library for using ANSI escape codes to format the console output. https://github.com/fusesource/jansi, 2021.

[20] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. {GREYONE}: Data flow sensitive fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2577–2594, 2020.

[21] google. A multi-language, cross-platform library of cryptographic APIs. https://github.com/google/tink, 2021.

[22] google. A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris, 2022.

[23] google. Continuous Fuzzing Framework for Open Source Software. https://github.com/google/oss-fuzz, 2022.

[24] Manel Grichi, Mouna Abidi, Fehmi Jaafar, Ellis E Eghan, and Bram Adams. On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (jni). *IEEE Transactions on Reliability*, 70(1):428–440, 2020.

[25] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 25–28. IEEE, 2018.

[26] Mary Jean Harrold, Gregg Rothermel, and Alex Orso. Representation and analysis of software. *Lecture Notes*, 2005.

[27] igraph. A C library for creating, manipulating and analysing graphs. https://github.com/igraph/igraph, 2022.

[28] ISC. A Classic, full-featured and mostly standards-compliant DNS. https://gitlab.isc.org/isc-projects/bind9, 2022.

[29] javaparser. A set of libraries implementing a Java 1.0 - Java 15 Parser. https://github.com/javaparser/javaparser, 2022.

[30] jhy. Java HTML Parser. https://github.com/jhy/jsoup, 2022.

[31] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. Rulf: Rust library fuzzing via api dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 581–592. IEEE, 2021.

[32] jna. Java Native Access. https://github.com/java-native-access/jna, 2020.

[33] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2511–2513, 2017.

[34] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[35] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[36] Legrandin. An self-contained Python package of low-level cryptographic primitives. https://github.com/Legrandin/pycryptodome, 2018.

[37] Wen Li, Haipeng Cai, Yulei Sui, and David Manz. PCA: Memory leak detection using partial call-path analysis. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Tool track*, pages 1621–1625, 2020.

[38] Wen Li, Li Li, and Haipeng Cai. On the vulnerability proneness of multilingual code. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[39] Wen Li, Na Meng, Li Li, and Haipeng Cai. Understanding language selection in multi-language software projects on github. In *IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings*, pages 256–257, 2021.

[40] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. PolyCruise: A cross-language dynamic information flow analysis. In *31st USENIX Security Symposium*, pages 2513–2530, 2022.

[41] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. PATA: Fuzzing with path aware taint analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 154–170, 2022.

[42] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. {FANS}: Fuzzing android native system services via automated interface analysis. In *29th USENIX Security Symposium*, pages 307–323, 2020.

[43] LLVM. LibFuzzer: A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2020.

[44] luben. JNI bindings for Zstd native library. https://github.com/luben/zstd-jni, 2021.

[45] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[46] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[47] mozilla. An allowed-list-based HTML sanitizing library. https://github.com/mozilla/bleach, 2022.

[48] msgpack. An efficient binary serialization format. https://github.com/msgpack/msgpack-python, 2021.

[49] M.Zalewski. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2014.

[50] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187. IEEE, 2019.

[51] ninia. Java Embedded Python. https://github.com/ninia/jep, 2018.

[52] OK.ru. A library for building high performance Java servers. https://github.com/odnoklassniki/one-nio, 2020.

[53] OWASP. A JSON encoder in Java. https://github.com/OWASP/json-sanitizer, 2017.

[54] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium*, pages 729–743, 2018.

[55] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[56] pillow. Python Imaging Library. https://github.com/python-pillow/Pillow, 2019.

[57] pybind. Seamless operability between C++11 and Python. https://github.com/pybind/pybind11, 2022.

[58] pycurl. A Python Interface To The cURL library. https://github.com/pycurl/pycurl, 2022.

[59] pydata. A a collection of fast NumPy array functions. https://github.com/pydata/bottleneck, 2020.

[60] pygments. A generic syntax highlighter written in Python . https://github.com/pygments/pygments, 2022.

[61] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[62] Rust Fuzzing Authority. Rust-fuzz. *Available online at: https://github.com/rust-fuzz*, 2020.

[63] simplejson. A simple, fast, complete, correct and extensible JSON. https://github.com/simplejson/simplejson, 2022.

[64] SQLAlchemy. The Python SQL Toolkit and Object Relational Mapper. https://github.com/sqlalchemy/sqlalchemy, 2022.

[65] Robert Swiecki. Honggfuzz. *Available online at: http://code.google.com/p/honggfuzz*, 2016.

[66] tree-sitter. JavaScript and JSX grammar for tree-sitter. https://github.com/tree-sitter/tree-sitter-javascript, 2022.

[67] tree-sitter. Ruby grammar for tree-sitter. https://github.com/tree-sitter/tree-sitter-ruby, 2022.

[68] tytso. The filesystem utilities for use with the ext2 filesystem. https://github.com/tytso/e2fsprogs, 2022.

[69] ultrajson. An ultra fast JSON encoder and decoder. https://github.com/ultrajson/ultrajson, 2020.

[70] urllib3. A HTTP client for Python. https://github.com/urllib3/urllib3, 2022.

[71] Vasudev Vikram, Rohan Padhye, and Koushik Sen. Growing a test corpus with bonsai fuzzing. In *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2021.

[72] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *NDSS*, 2021.

[73] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, 2018.

[74] yaml. A full-featured YAML processing framework for Python. https://github.com/yaml/pyyaml, 2022.

[75] Haoran Yang, Wen Li, and Haipeng Cai. Language-agnostic dynamic analysis of multilingual code: Promises, pitfalls, and prospects. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Ideas, Visions and Reflections*, 2022.

[76] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[77] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[78] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th {USENIX} Security Symposium*, pages 2307–2324, 2020.

[79] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th {USENIX} Security Symposium*, pages 2255–2269, 2020.

[80] zxing. A multi-format 1D/2D barcode image processing library. https://github.com/zxing/zxing, 2022.

A More on PolyFuzz Implementation

Language-specific analysis and instrumentor. For each language, we implemented an SAIR translator and an instrumentor. Specifically, the implementation for C works in three primary steps in one LLVM pass [35]: (1) Translates the LLVM intermediate representation to SAIR following the syntax described in §3.2.1. (2) Use *IGC* to compute instrumentation guidance based on the SAIR. (3) Instruments the dynamic tracing APIs defined in the *DynTrace* library according to the guidance computed. The implementation for Java is similar, but on top of Soot [34] with JNI wrappers of *DynTrace* and *IGC*. For Python, the SAIR translator and instrumentor are implemented separately: we developed a static parser and SAIR translator based on AST, and a dynamic instrumentor using Pybind [57]. Overall, adding support for a new language is lightweight, as all complex algorithmic implementations are already provided in the three common C libraries mentioned.

IGC. In IGC, we implemented intraprocedural control and data flow analysis [26] based on the SAIR of the given program. As the output of IGC, each instrumentation guidance is a value pair <block-id, statement-id> for a given function. Moreover, the implementation of IGC is thread-safe to support parallel running of compiler or program analysis frameworks (e.g., LLVM [35] and Soot [34]).

Dynamic tracing (DynTrace). We implemented this library in C with three primary functionalities: (1) an API for initializing the shared-memory byte map for coverage computation in AFL++; (2) an API for initializing the shadow event queue for caching covered branch variables during SASG. (3) APIs for tracing dynamic events (e.g., block information, branch variables). All these APIs can be invoked by the language instrumentors directly or through a wrapper of corresponding language interfaces (e.g., a JNI wrapper for Java) and then inserted into the fuzzing targets.

SASG and AFL++. For better fuzzing efficiency, SASG and AFL++ run in parallel most of the time; also, SASG does not generate seeds all at once—instead, every time it generates 8K seeds, it informs AFL++ to load and fuzz. During the adaptive model selection, the regression accuracy is measured (during model validation) as the mean distance between predicted and ground-truth seed-block values. When this accuracy drops below 80%, the regression is considered a failure.