CONFETTI: Amplifying Concolic Guidance for Fuzzers

James Kukucka[1], Luís Pina[2], Paul Ammann[1], and Jonathan Bell[3]

[1]George Mason University, Fairfax, VA USA

[2]University of Illinois Chicago, Chicago, Illinois USA

[3]Northeastern University, Boston, MA USA

jkukucka@gmu.edu,luispina@uic.edu,pammann@gmu.edu,j.bell@northeastern.edu

Abstract

Fuzz testing (fuzzing) allows developers to detect bugs and vulnerabilities in code by automatically generating defect-revealing inputs. Most fuzzers operate by generating inputs for applications and mutating the bytes of those inputs, guiding the fuzzing process with branch coverage feedback via instrumentation. Whitebox guidance (e.g., taint tracking or concolic execution) is sometimes integrated with coverage-guided fuzzing to help cover tricky-to-reach branches that are guarded by complex conditions (so-called "magic values"). This integration typically takes the form of a targeted input mutation, *e.g.,* placing particular byte values at a specific offset of some input in order to cover a branch. However, these dynamic analysis techniques are not perfect in practice, which can result in the loss of important relationships between input bytes and branch predicates, thus reducing the effective power of the technique. We introduce a new, surprisingly simple, but effective technique, *global hinting*, which allows the fuzzer to insert these interesting bytes not only at a targeted position, but in any position of any input. We implemented this idea in Java, creating Confetti, which uses both targeted and global hints for fuzzing. In an empirical comparison with two baseline approaches, a state-of-the-art greybox Java fuzzer and a version of Confetti without global hinting, we found that Confetti covers more branches and finds 15 previously unreported bugs, including 9 that neither baseline could find. By conducting a post-mortem analysis of Confetti's execution, we determined that global hinting was at least as effective at revealing new coverage as traditional, targeted hinting.

ACM Reference Format: James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510628

1 Introduction

Software is at the core of critical electronic systems. To avoid introducing faults, which can lead to significant errors and security vulnerabilities, developers test their applications before deployment by generating diverse inputs that exercise as many behaviors as possible, attempting to catch bugs and vulnerabilities before they escape to the wild. Unfortunately, manual testing only goes so far towards generating diverse (and unexpected) inputs. Many recent advances in greybox fuzzing, such as the popular American Fuzzy Lop (AFL) [81], AFL++[38], libFuzzer [55], and hongfuzz [44], are based on *coverage-guided fuzzing*. Coverage-guided fuzzers use branch coverage as feedback to guide mutation of a set of manually provided "seed" inputs towards new inputs that explore new program paths. These fuzzers might execute thousands of inputs per second, but are unlikely to generate inputs that satisfy highly constrained branches which require some so called "magic bytes".

A complementary approach, *concolic execution*, discovers those magic values by recording exactly which input bytes are used in which branches in the program's execution). Then, with the aid of an SMT solver, the concolic execution engine generates inputs that force a different branch choice [45, 72, 82]. Prior work has observed that full-blown concolic execution is often unnecessary to handle typical magic byte comparisons, turning instead to dynamic taint tracking [33, 40, 69]. Dynamic taint tracking is an analysis that associates taint tags with values, and then propagates those tags during program execution such that when a new value is derived (through data flow) from a tainted value, that same taint tag is associated with the new value.

While taint tracking-guided fuzzers like VUzzer [69], Angora [33] and BuzzFuzz [40] have been shown to be more effective than a typical greybox fuzzer, we believe that they have only begun to leverage the power of taint tracking in fuzzing. In particular, taint tracking can only guide the fuzzer to explore branches for which there is a *dataflow relationship* between the branch predicate and the input bytes. Consider the code snippet in Listing 1, in which the input strings `s1` and `s2` are compared against some particular string, with that comparison stored into a boolean variable.

```
1 public void magic(String s1, String s2){
2   boolean v1 = s1.equals("abc");
3   boolean v2 = s2.equals(s1.concat("def"));
4   if(v1 && v2)
5     throw new IllegalStateException(); //Bug
6 }
```

Listing 1: Example code in which taint tags from inputs `s1` and `s2` do not flow to a branch that they indirectly control.

Ideally, the taint tracking tool could report to the fuzzer that to cover the true side of the branch on line 4, the fuzzer must mutate `s1` and `s2` (and even better, to generate the concrete values `abc` and `abcdef`, respectively). However, the taint tracking tool will not report any relationship between the input and the branch on line 4 because `v1` is control-dependent on `s1`, but not data-dependent (and similarly, between `s2` and `v2`). Even worse: while in this example, there is a dataflow relationship between the input strings `s1` and `s2` and the magic strings `abc` and `abcdef`, in real code, the taint

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00

https://doi.org/10.1145/3510003.3510628

tags on s1 or s2 might also be lost through implicit flows. For example, one common pattern is to build a map from input strings to a tokenized representation of each string — if the same input string is encountered more than once, the parser returns the same tokenized version of the string, effectively losing track of the input.

While some dynamic taint tracking tools do support "control flow propagation," which would detect this relationship, these analyses have too many false positives to be useful in practice [34, 47]. How else can we help the fuzzer to explore this branch? One typical approach is to simply scrape the application binary for all strings, creating a dictionary of interesting strings to use when fuzzing (in this case, abc and def). Unfortunately, this trick only works if the magic values are statically defined in the codebase — values generated dynamically will not be included in the dictionary. In this case, because s2 must be the value abcdef, the dictionary will not help the fuzzer explore this branch.

This paper presents Confetti, (CONcolic Fuzzer Employing Taint Tracking Information), a system that combines fuzzing with taint-tracking and concolic execution. Confetti amplifies the reach of concolic guidance, allowing the fuzzer to effectively generate inputs that explore branches like the one in Listing 1, *and* longer, more complex examples where taint tags quickly become lost through implicit flows. Our key insight is that the precise targeting of past taint tracking-guided greybox fuzzers unnecessarily restricts the fuzzer's ability to reveal tricky-to-reach branches. As with state-of-the-art fuzzers, Confetti executes each input in its population with taint tracking, collecting constraints on the input bytes. Confetti can generate new coverage-revealing inputs through concolic execution by negating and solving those constraints, in the style of existing work [33, 35, 63, 72, 79, 80].

Confetti's novel approach to guide the fuzzer, *global hinting*, is based on the insight that although taint tags might be lost for parts of an input, magic values derived for *other* parts of the input can be re-targeted and applied elsewhere. When Confetti finds that a part of the input flows into a comparison with a dynamically computed value, Confetti records that value as a global hint. We create and evaluate a new fuzzing mutation, which inserts global hints *anywhere* in *any* input — not only at the targeted location of the specific input from which the hint was derived.

We evaluate the efficacy of this new mutation strategy, considering both system-level metrics (*i.e.,* branches covered and bugs found) and unit-level metrics (*i.e.,* mutation success rate). Our results clearly demonstrate that global hinting is roughly as effective in revealing new coverage as traditional, targeted hinting, and most importantly, that this strategy reveals *different* coverage and bugs that *could not be reported by using targeted hinting alone.* In our evaluation, the baseline JQF-Zest fuzzer detected 11 bugs, whereas Confetti with only targeted hints detected 16, and Confetti with both global and targeted hints detected 25 bugs. Our open-source implementation of Confetti represents a significant improvement in fuzzer technology for JVM-based software, providing benefits to software engineering researchers inventing new fuzzing approaches and to professional software engineers searching for bugs in their software. While our implementation is limited to a single language (Java) and a single greybox fuzzer (Zest), we believe that our results are compelling enough to have a significant impact on the field of software engineering, warranting future work exploring global hinting in other fuzzing domains.

The key contributions of this paper include:

- A new approach to combine concolic execution and taint tracking with fuzzing: *global hinting*.
- An open source implementation of Confetti for Java, which combines traditional, targeted hinting with our novel global hinting strategy [51, 52].
- An evaluation of Confetti, demonstrating the efficacy of its novel global-hinting-based guidance over a baseline state-of-the-art greybox Java fuzzer (Zest), and against a baseline version of Confetti without global hinting.

2 Background

Before describing how Confetti effectively guides a greybox fuzzer using whitebox information, we first briefly summarize greybox fuzzing, and in particular, parametric greybox fuzzing. Consider fuzzing an application that takes XML files as input. Figure 1 shows two fuzzing loops: one that represents the behavior of a traditional coverage-guided fuzzer like AFL [81] or libFuzzer [55] (blue line), and one that represents the behavior of a parametric fuzzer like Crowbar [37], FuzzChick [53] or JQF-Zest [66] (orange line). The traditional fuzzer (blue) executes a loop, where it starts with some (well-formed) seed input, selected from a pool of seeds. The fuzzer then uses a *mutator* to transform that input (typically using an evolutionary algorithm). Then, the fuzzer executes the new input and captures branch coverage that may bias the evolutionary algorithm (mutator) on future executions of the fuzzing loop. If the new input is deemed *interesting* — typically defined as revealing coverage of a new branch, or greatly increasing the hit counts of those already covered — then the input is saved into the fuzzer's population, to be selected again later for further fuzzing.

However, in the case of the traditional coverage-guided fuzzer (blue line), the mutator is unaware of the input syntax expected by the system under test, so most of the generated inputs are likely to have a shallow reach in the code. That is, most of these inputs either fail some early stage syntactic parsing (*e.g.,* the XML fragment <xml><^xml>), or at best, find a bug in that syntactic parser (perhaps <xml><^xml> causes a crash).

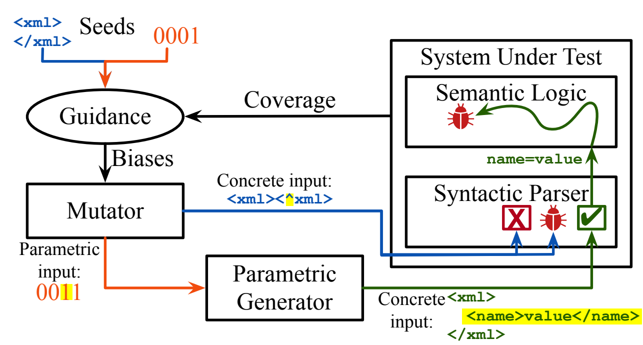


Figure 1: Comparing a fuzzing loop for a traditional coverage-guided fuzzer (blue) and parametric fuzzer (orange). The traditional fuzzer uses an evolutionary algorithm to mutate concrete program inputs directly. The parametric fuzzer mutates the *parametric input* — the sequence of decisions made by a generator function that ultimately creates concrete inputs.

In contrast, the parametric fuzzer (orange and green lines in Figure 1) has a seed pool that consists of *parametric inputs*, which are the sequences of decisions made by the parametric generator that result in some input. Whereas property-based testing tools employ *random* generation, parametric fuzzers guide the generation of new inputs by controlling each "random" decision made by the generator. The parametric input `0001`, in this example, represents the set of decisions made by the generator function to create the concrete input `<xml></xml>`. The parametric fuzzer also uses a mutator to transform a seed input, but operates on this parametric input (orange line). By mutating parametric inputs, corresponding mutations occur at the *object* level and not at the input byte level. A one-bit mutation to `0011` might result in a more semantically interesting change to the concrete input, creating `<xml><name>value</name></xml>` in the example. Hence, the key insight behind parametric fuzzing is that the *structure* of inputs is often more constraining than the set of values inserted into that structure. Prior experiments show that parametric fuzzing is more effective when compared to traditional property testing [53] and to traditional coverage-guided fuzzing of Java programs [66]. Since we target Java applications, we chose to integrate CONFETTI with a parametric fuzzer.

3 CONFETTI

CONFETTI generates complex inputs that can expose hidden bugs in a program's logic by using concolic execution and taint tracking as forms of guidance for parametric fuzzing. The traditional approach to integrate whitebox guidance with fuzzing is to provide *targeted* guidance, instructing the fuzzer to place particular bytes at a particular location in a particular input. We refer to each such suggestion as a "targeted hint." CONFETTI employs targeted hints, but also introduces the notion of *global* hints, which allow CONFETTI to overcome the inadequacies of dynamic taint tracking. Although our approach is language-agnostic, we implement CONFETTI in Java and target applications written in languages that target the JVM such as Java, Scala, Kotlin, Groovy, and Clojure.

3.1 Example

Before examining CONFETTI's architecture in detail, shown in Figure 3, we provide a brief example to demonstrate how CONFETTI works in tandem with the fuzzer to generate concrete inputs that explore otherwise hard-to-reach code. Figure 2 shows CONFETTI's hinting process to bias an example XML generator — in this example, it is the `generateXML()` function called in line 1. As stated previously, a parametric input can be thought of as a series of choices that are made — we have color-coded distinct choice points with the XML generator, along with their associated parametric input and resulting, concrete values. Function `generateString` is a black box that consumes a parametric input and generates a string.

We present the *minimal* set of program executions that reaches line 6 — but in practice, CONFETTI applies a variety of mutators to support both targeted and non-targeted hints (described further in Section 3.5). Initially, the generator creates a random seed input (① in Figure 3) — Input #1, which randomly selects the string "groupID" (perhaps from a dictionary), and the random boolean `false`. Being the first input of the fuzzing run, this is guaranteed to obtain new coverage — namely lines 2 and 3. This input is deemed "interesting" by the fuzzer, as it leads to increased branch coverage, and is sent to the CONFETTI Coordinator for whitebox analysis (② in Figure 3).

```
1 XMLDocument doc = parse(generateXML());
2 if(!"expected".equals(doc.element(0).name()))
3    throw new Error();
4 if(!"version".equals(doc.element(0).attr(0).name()))
5    throw new Error();
6 // interesting logic we want to fuzz follows bellow
```

(a) Example code to fuzz; the immediate goal is to pass both branches

```
String generateXML(){
  String tagName = generateString();
  if(generateBoolean())
    return "<"+tagName+ " "+
      generateString() +"='"+
      generateString() +"'></"+tagName+">";
  return "<"+tagName+"></"+tagName+">";
}
```

(b) Simple parametric XML tag generator that uses 4 random choices

#	Source	Parametric Input	Generator Choices	New Coverage
1	seed	0101101 0	"groupID" false	5
2	mutation	0101111 0	"package" false	-
3	hint	1001011 0	"expected" false	6
4	mutation	1001011 1 011010 01101	"expected" true "groupID" "A"	7, 8
5	hint	1001011 1 011111 01101	"expected" true "version" "A"	9

(c) Visualization of Fuzzer Execution with hints. The first input is generated randomly, later inputs are *mutated* by the fuzzer, or *hinted* by CONFETTI (hint underlined). CONFETTI generates hints from coverage-revealing inputs only.

Figure 2: Example Fuzzer Execution. A parametric fuzzer can easily create an input that is parsable by the black-box function on line 1, but may struggle to reach line 6 due to lines 2 and 4. As the parametric fuzzer records the sequence of random choices made in the generator (a), and mutates those choices to generate new inputs, CONFETTI proposes values for some of those choices, thus guiding fuzzer through these branches with ease.

The CONFETTI Coordinator dispatches the input to a whitebox analysis process ③ to perform taint-tracking and collect constraints, which are ④ returned to the CONFETTI Coordinator, which ⑤ negates and solves them, ⑥ possibly resulting in new parametric inputs. Meanwhile, ① the fuzzer mutates inputs independently, changing "groupID" to "package" in Input #2.

Once the fuzzer decides to mutate a new input, it contacts the CONFETTI Coordinator for hints, which the ⑦ CONFETTI Coordinator returns in the form of the modified parametric input derived from Input #1, hinting the first generated string should be "expected" (derived from taint tracking between the input string and the `String.equals` call). With the hint, Input #3 results in new coverage on line 4 (an exception is thrown immediately upon hitting line 4 due to the absence of attributes) — Input #3 is then sent to the CONFETTI Coordinator for whitebox analysis. Meanwhile, the fuzzer mutates Input #3 into #4 by changing the boolean choice to `true`, followed by a random string for an attribute name and value. Input #4 results in new coverage on line 5, and is also sent to the CONFETTI Coordinator. Input #5 is generated in a similar manner to Input #3, using CONFETTI's hint mechanism for strings by instrumenting as the call to `attr` to obtain the hint "version." While the parametric fuzzer *could* create these inputs by chance, it becomes a certainty with CONFETTI.

3.2 Architecture

Figure 3 shows a high-level overview of CONFETTI's architecture. CONFETTI consists of three key processes that run in coordination: *(1) the fuzzer* — responsible for input generation and execution of

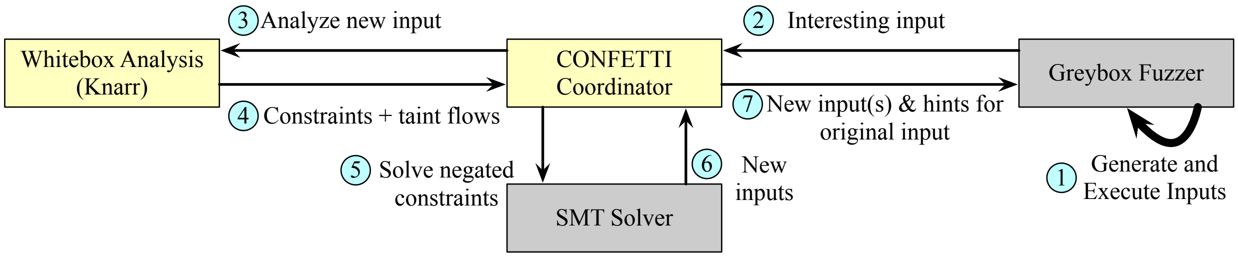


Figure 3: Overview of CONFETTI's approach to provide whitebox hinting to a greybox fuzzer. The fuzzer repeatedly generates and executes inputs, considering any hints that hit has received so far to guide its generation ①. For each input that the fuzzer creates that is deemed interesting for generating new coverage, it ② sends that newly generated input to the CONFETTI coordinator, which in turn ③ sends the newly generated input to the KNARR client in order to ④ perform a whitebox analysis and collect constraints. If there are constraints, the CONFETTI coordinator ⑤ negates constraints and sends them to the SMT solver to ⑥ generate new inputs. Regardless of the SMT result, the CONFETTI coordinator also extracts taint tracking-derived hints from string comparisons in the constraints. The coordinator ⑦ returns the hints, and any new SMT-derived input(s) to the fuzzer, and the cycle continues.

the target program, *(2) the KNARR Process* — responsible for dynamic taint tracking and constraint collection on interesting inputs, and *(3) the CONFETTI Coordinator* — responsible for transmitting inputs between the two aforementioned processes and for using an SMT solver to feed concolic execution. This design has two main benefits. First, it uses separate processes for components that execute the program under test with different (incompatible) instrumentations: collect taint and constraints (KNARR), or simple branch coverage (fuzzer). Second, it allows the whitebox analysis to take place *without* delaying the fuzzer, which continues to mutate inputs unimpeded by constraint collection and analysis. The CONFETTI Coordinator thus acts as a broker for inputs being passed between KNARR and the fuzzer, receiving interesting inputs from the fuzzer and forwarding them to KNARR for further analysis. Similarly, the CONFETTI Coordinator manages constraints that are computed by KNARR, negating and solving them in the SMT solver.

Similar to prior work such as Angora [33], the CONFETTI Coordinator is not notified of each mutation/execution that the fuzzer performs, but instead only of each input that the fuzzer finds interesting (*e.g.,* each input that reaches new coverage). Focusing on interesting inputs has two important consequences. First, KNARR executes a small number of inputs and does not lag behind the fuzzer. Second, this mitigates the path explosion problem since KNARR executes (concretely) paths known to reach new coverage, and the fuzzer ultimately selects which hints to take. In this way, CONFETTI leverages the speed of the fuzzer at generating inputs and the power of the whitebox analysis.

Our implementation of CONFETTI extends the JQF+Zest parametric fuzzer [64–66] and extends our PHOSPHOR dynamic taint tracking engine [27, 28] for constraint collection. Our prototype interfaces uses the Green library [75] as a bridge to constraint solvers, allowing it to be agnostic of the solver used. In practice, we use the mature Z3 theorem prover [36], which worked well in our experiments. Future work might consider other constraint solvers, perhaps using newer Java APIs like JavaSMT3 [23].

3.3 Knarr: Collecting Whitebox Guidance

KNARR uses *dynamic taint tracking* to trace how each byte of the parametric input flows through the generator into a generated concrete input, and then through the application under test. Dynamic taint tracking is an automated analysis that allows tools to *taint* some variable(s), and then, at any point in the program execution, identify if a variable is derived through dataflow from that original, tainted input. KNARR instruments the system under test (including the generator that drives the application) to perform this analysis.

Recall from Section 2 that a parametric fuzzer represents each input as a series of random choices consumed by a generator program. To taint the generated input, we modify the fuzzer to taint each byte of the parametric input that is consumed by the generator. By doing this, we are able to propagate taint tags through the parametric input to the concrete generated input and beyond. Most generators require no changes, the only modifications that CONFETTI may require to the generator are for string generators that selecting a random item from a pre-defined dictionary. For example, such a generator might have logic along the lines of `result = dict[choice % dict.length]`, where `choice` is a random integer and `dict` is a pre-defined list of strings. CONFETTI requires these generators to be rewritten to call a helper function, along the lines of `result = ConfettiHelper.stringFromList(choice, dict)`. This helper function will propagate the taint tag from `choice` to `result` (since array-indexing is an implicit flow), and will allow CONFETTI to decide to use a hint (which may not be defined in the dictionary) or to choose an item from the dictionary.

KNARR tracks taint tags for each variable, and for strings, tracks taint tags at a per-character level. KNARR tracks common string operations like `equals` and `startsWith` so that it can represent these operations to the solver. When executing the generator and the concrete input, KNARR records the taint tag of values used in branch predicates. KNARR sends this data to the CONFETTI Coordinator, that can then connect individual bytes in a given parametric input to conditions guarding branch edges not yet covered.

Instead of using a simple, traditional taint tag (of 'tainted' or 'not tainted'), KNARR enhances the taint tracking engine to build an abstract expression for each variable to use as the taint tag (again, a technique inspired by Angora [33]). For instance, given the code `int x = y + z` and assuming that `y` and `z` were tainted inputs, an off-the-shelf taint tracking tool would typically set `x`'s taint tag to be the union of `y` and `z`'s tags. Instead, KNARR tracks the abstract expression that generated the value (in this case, that $x = y + z$). In this way, `x`'s taint tag becomes the symbolic expression $y + z$.

When a tainted input reaches a branch, the taint tag of the branch condition is then the complete symbolic expression that relates the parametric input byte to the branch condition.

When Knarr detects tainted data being used in a branch, it adds the constraints in the tainted data to the current *path condition*. The path condition is thus the conjunction of all the constraints observed to control branches while executing one input. After executing each input, Knarr collects all constraints in the current path condition and sends them to the Confetti Coordinator, which uses those constraints to generate new inputs and hints for the fuzzer.

3.4 Confetti Coordinator and Hints

Using the constraints collected by Knarr, Confetti Coordinator derives three kinds of targeted hints: SMT solver-derived hints, string comparison-derived hints and character comparison-derived hints. Confetti Coordinator provides these to the fuzzer as targeted hints, and as explained in the following section, the fuzzer will derive a set of global hints from these targeted hints. Confetti Coordinator leverages an SMT solver in the style of *concolic execution* [42, 71] in order to generate new inputs that are likely to reveal new branch coverage. While in principle, Confetti Coordinator could attempt to negate and solve all unique branch conditions in order to attempt to explore all paths, in practice we found that concolic execution was most useful to target branches that could not be covered by the fuzzer. As Knarr executes inputs and collects path constraints, Confetti Coordinator keeps track of which branches have not been fully explored.

Confetti's concolic execution thread works by first selecting a branch to target — one that is not fully covered and whose predicate includes at least one value from the input. Then, Confetti selects one of the inputs that reaches the branch and negates the constraints applied by that branch's predicate. Confetti drops constraints from the input that occurred after this branch execution, since it might be unsatisfiable to retain them while also negating the target branch's constraints. Then, Confetti uses an SMT solver (Z3 [36]) to generate a new parametric input that takes the other side of the branch. If satisfiable, the solution is then translated into a new, hinted input that can be immediately executed by the fuzzer. If the solver deems the constraints to be unsatisfiable, or times-out, Confetti marks that combination of input and branch as "already tried" and moves on to the next target branch.

After attempting to generate inputs for all uncovered branches once, Confetti loops around to try each uncovered branch again, this time picking a new input. Confetti records solver-related statistics: how often a branch was targeted for solving how often each input was tried to solve for that branch, and the result of that solver call. Some branches may never be satisfiable, due to limitations in constraint tracking or solving (*e.g.,* usage of floating point operations), and perhaps become a waste of solver time. We found that most branches that could be solved for were often solved on one of the first few inputs attempted, and added a user-configurable threshold to blacklist particular branches that repeatedly were not satisfiable, defaulting to 50 attempts.

Since Confetti's goal is to provide *guidance* to a fuzzer (and not necessarily perform complete concolic execution), it also provides very lightweight, taint tracking derived hints to the fuzzer. Confetti extracts comparisons between input values and various string values, regardless of whether those comparisons control branches are covered. For each of these string comparisons, Confetti provides the fuzzer with a targeted hint to set the relevant bytes of the input to the value that was compared to.

Since Knarr tracks taint tags on each *character* of each string, it is also possible for Confetti to derive hints from comparisons between individual characters of strings. To mitigate the explosion of hints resulting from suggesting *every* possible character that a given input is compared against, Confetti limits the total number of character hints suggested for each targeted branch to 10.

3.5 Parametric Fuzzer Guidance

Confetti's core novelty over prior work is in how it *integrates* those results with the fuzzer. State-of-the-art fuzzers that integrate guidance from dynamic taint tracking and/or path constraint solving — like Angora [33], Driller [72], VUzzer [69] and others [35, 63, 79, 80] — provide *targeted* guidance to the fuzzer. For instance, taint tracking might be used to determine which bytes of the input control branches that are not yet covered, and then the fuzzer might be guided to generate a particular input to cover that branch. Confetti uses several targeted hinting strategies based on prior work in addition to its novel, *global* hinting strategy.

When mutating an input, Confetti extends the fuzzer with the following new mutations: 1) Apply a single targeted hint, 2) Apply multiple targeted hints, simultaneously, or 3) Perform normal mutation, which might apply global hints

Targeted hints represent the state-of-the-art approach to integrate taint tracking and constraint solving with fuzzing: if Knarr determines that there is a particular value that should be tried at a particular position in an input, then that value is applied to that offset. Targeted hints are always applied *without* further mutation of the input, since the hints were collected on the original input being mutated, an arbitrary change to the input might invalidate the usefulness of those hints. When the fuzzer selects an input for mutation, and there are targeted hints that have not yet been applied, with a coin flip, one of those targeted hints is applied. After an input is selected for mutation repeatedly, eventually all targeted hints will be tried, and then this mutation will no longer be available for this input. If a single hint isn't applied, then the fuzzer might apply multiple targeted hints simultaneously. In either case, Confetti inserts instructions in the input to use the hinted value, rather than whatever value would have otherwise been chosen by the generator at that targeted position.

Each time that a targeted hint is applied to an input, that hint value is saved in a global hint set, enabling Confetti's powerful global hinting mechanism. This global hint set tracks all strings that any input string was compared to during the fuzzing campaign. At any call in the generator that *could* consume a targeted hint, we add a coin flip to determine whether the global hint set should be used, or the generator's normal logic should be used. By mutating the bits that control this decision, the fuzzer can control the application of global hints at each position. In our evaluation, we found that this seemingly simple strategy was very effective at generating new, coverage-revealing inputs and in revealing new bugs.

A key aspect of Confetti's hinting implementation is that it ensures that hints are *inheritable*: if an input with targeted or global hints is deemed useful, saved, and fuzzed later, assuming that the choices to generate those hinted values aren't mutated, then the same hints will be applied in the same position. This allows the fuzzer to make progress towards generating increasingly more complex inputs by stacking multiple hints together. We did not perform any hyper-parameter tuning to optimize the probabilities of applying hints based on their overall performance, although in Section 4.3 we report on the success rate of each mutation strategy.

4 Evaluation

In order to empirically evaluate Confetti and, in particular, its novel global hinting strategy, we measured its effectiveness across a suite of benchmark programs. Our evaluation is primarily focused on answering the following research questions:

RQ1: How does Confetti compare to the baseline fuzzers in terms of branches explored?

RQ2: Does Confetti find bugs that the baseline fuzzers cannot?

RQ3: How useful is each of Confetti's hint strategies for discovering new coverage-revealing inputs?

RQ4: Can inputs with Confetti's global hints be replaced with statically derived values, and still yield the same coverage?

We evaluate Confetti in comparison to the state-of-the-art parametric fuzzer JQF-Zest [66] and use the same suite of benchmark programs, given that we built Confetti on top of JQF-Zest. Where possible, we used the latest version of the target software that still contained the bugs detected by JQF-Zest in the original work. Following best practices, we study both Confetti's ability to explore program branches (*e.g.,* coverage) in comparison to JQF-Zest, and its ability to find new and previously-known bugs [50].

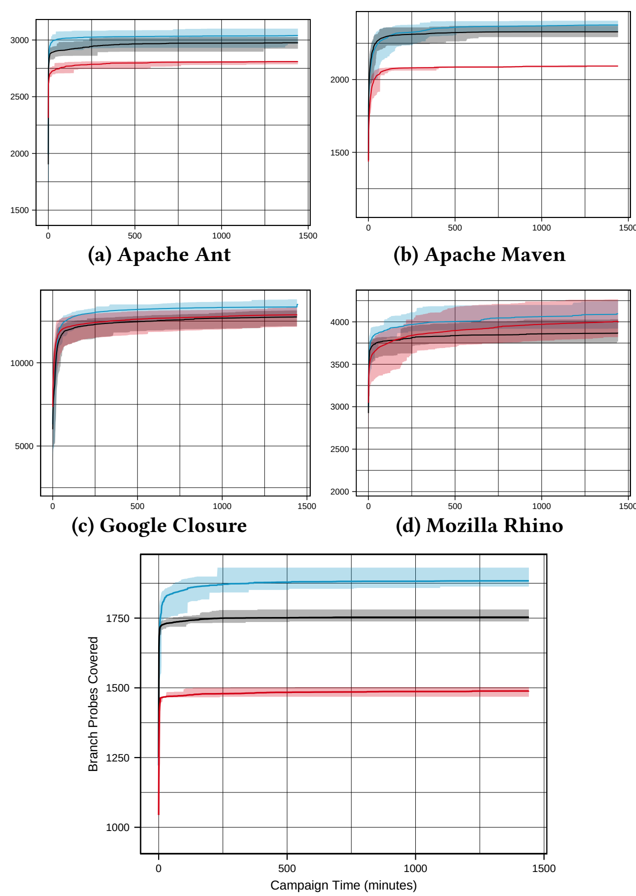
In order to precisely evaluate the efficacy of Confetti's global hinting strategy, we also evaluate Confetti's coverage and fault finding ability in comparison to a baseline Confetti$_{tgt}$, which is exactly the same version of Confetti, but with global hints disabled. However, there may be a variety of confounding factors that also impact Confetti$_{tgt}$'s overall performance in this head-to-head evaluation. For instance, the fuzzer's scheduling algorithm (that allocates mutation time to and chooses the mutations to apply on inputs) likely interacts with factors that influence the coverage of each input — like hints. To isolate the impact of global hints, we also analyzed each of the coverage-revealing inputs that Confetti generated, looking to determine whether or not those inputs could have been generated without global hints.

We conducted all of our experiments on Amazon's EC2 infrastructure, using "r5.xlarge" instances with 4 3.1Ghz Intel Xeon Platinum 8000 CPUs and 32 GB of RAM, running Ubuntu 16.04 "xenial" and JDK 1.8.0_241. Following best practices, we conducted each experiment for 24 hours and repeated this 20 times averaging the results [50]. The input generators used in our evaluation are extensions of the open source input generators that were published by the JQF-Zest authors within JQF itself [64]. The modifications made to the generators for constraint tracking and hinting are minimal, amounting to approximately two lines of code in the XML document generator, approximately four lines of code in the JavaScript code generator, and approximately ten lines of code in the Java class file generator. We also modified the Maven pom.xml generator — the code provided by the JQF-Zest authors was misconfigured, and hence unable to generate high-coverage pom.xml files. Our modification to the Maven pom.xml generator was merged into the upstream codebase [26]. Otherwise, we used the generators as provided by the JQF-Zest authors without modification.

4.1 RQ1: Evaluating Fuzzer Coverage

Most fuzzers (including JQF-Zest and AFL) consider coverage of *all* code, in both the application's code and its libraries, to determine which inputs to save, since an input that covers new library code might be "closer" to covering new application code. Following the methodology of Padhye *et al.* 's JQF-Zest [66], we analyze and report coverage overall, and also for application code specifically.

Figure 4 visualizes the branch coverage of *all* code (not only the system under test) of each of the 20 executions of each of the benchmark programs for each fuzzer during the duration of each 24-hour campaign. The solid line represents the average coverage across each of the 20 executions, and the shaded area represents the complete range of coverage. We also calculated the *total* branch coverage for each fuzzer over all of its 20 runs, this time using the standard code coverage tool JaCoCo [59], and reporting only branches in the program under test covered by any input from any



(a) Apache Ant **(b) Apache Maven** **(c) Google Closure** **(d) Mozilla Rhino** **(e) Apache BCEL**

Figure 4: Rate of new branch discovery for each fuzzer, JQF-Zest shown in red, Confetti in blue and Confetti-NoGlobalHints in black. The solid line shows the average coverage across all 20 runs, and the shaded area shows the entire range. All charts have the same axis labels as BCEL.

Table 1: Summary of results for RQ1 and RQ2: branch coverage and bugs found. Coverage in this table includes *only* coverage of application code (no library coverage). Total branches shows the number of branches considered; branch coverage is shown aggregated across all 20 runs for CONFETTI, the baseline Java fuzzer JQF-Zest, and CONFETTI_{tgt} (CONFETTI with targeted hints but without global hints).

Benchmark Program (Version)	Total Branches	Total Branch Coverage			Bugs Found		
		CONFETTI	JQF-Zest	CONFETTI_{tgt}	CONFETTI	JQF-Zest	CONFETTI_{tgt}
Apache Ant [6] (1.10.2)	23,361	872	859	871	1	1	1
Apache Maven [11] (3.5.2)	5,858	857	821	853	0	0	0
Apache BCEL [10] (6.2)	6,220	1,421	1,361	1,423	5	2	3
Google Closure [12] (20190415)	49,602	11,458	10,545	10,640	15	4	8
Mozilla Rhino [19] (1.7.8)	25,035	3,744	3,757	3,534	4	4	4

of the 20 runs. Table 1 shows the total branch coverage of each fuzzer, along with the number of branches considered for coverage.

For all fuzzing targets, CONFETTI's average branch coverage surpassed that of CONFETTI_{tgt}, which surpassed that of JQF-Zest. Rhino's comparison graph is much tighter than the other graphs, with a great deal of variance for both CONFETTI and CONFETTI_{tgt} when compared with that of JQF-Zest— and the maximum coverage of JQF-Zest was greater than CONFETTI. Digging deeper into Rhino, we can see from Table 1 that, in total, JQF-Zest explored 13 more application branches than CONFETTI. This variance is likely due to the additional choices that CONFETTI introduces in the generators — namely by increasing the size of the global dictionary, or by having several hints to choose from at targeted byte positions. With further (and longer) trials, we suspect that this variation (and diversity) may help CONFETTI to ultimately achieve higher coverage than JQF-Zest. We also believe that future work could improve the efficacy of the generator for Rhino. For Maven, BCEL and Closure, CONFETTI's improvement in branch coverage over JQF-Zest was quite notable.

It is interesting to note that in the case of coverage BCEL, our baseline without global hints (CONFETTI_{tgt}) slightly outperformed CONFETTI. However, CONFETTI outperformed CONFETTI_{tgt} both in terms of total coverage (Figure 4) and bugs found — supporting our hypothesis that global hints are a useful strategy for combining concolic guidance with greybox fuzzing. We believe the limited variability in BCEL is due to JQF-Zest's Java class file generator, which we deliberately did not modify. In particular, it generates method bodies with very few instructions, greatly limiting the chances of exercising complex behavior in the target. Nonetheless, CONFETTI still outperforms JQF-Zest on this target, which shows that even with restrictive generators, CONFETTI still improves performance.

4.2 RQ2: Bugs Found

We analyzed each failure detected in all twenty, 24-hour runs, and reported each unique program crash as a bug in Table 2. In order to de-duplicate bugs, we utilize a heuristic of examining the first 5 lines in a stack trace to identify a unique bug, as well as manual analysis after applying this heuristic. This methodology clusters more bugs together than prior work of stack hashing [50], as the higher levels of the stack tend to isolate the locality of a particular bug. Using this methodology, we replicated the same 10 bugs that Padhye *et al.* reported in JQF-Zest (with even greater frequency in some cases), plus one additional bug in Closure, likely found due to performance improvements that we made to JQF-Zest (described in Section 5).

Of those 11 bugs that JQF-Zest found, CONFETTI found all but one in its twenty 24-hour runs (Issue B1 in the table). Again, we attribute this to additional choices that CONFETTI introduces in the generator, which clearly can result in a diversity of paths explored. This is evident in BCEL particularly, as CONFETTI finds four additional bugs that JQF-Zest does not. The addition of these choices makes bugs that are repeatable with low frequency even less likely to be triggered within a single run. We suspect that tuning the rate at which hints are selected could increase the likelihood that CONFETTI detects bugs like this, but leave such investigation for future work.

Of those 25 bugs that CONFETTI found, 16 (64%) were previously unknown, the rest had been found previously by JQF-Zest or others. Table 2 shows that, of the bugs that CONFETTI detects, there is a clear range of detectability, with some bugs detected on most fuzzing runs, and four detected at the 5% (*i.e.,* 1/20) level. This distribution supports our hypothesis that supplying global hints

Table 2: Bug detectability rate, from 20 executions of each fuzzer. If multiple unique bugs had the same repeatability rates, they are included in the same row (C14, C15, and R1, R2, R3, R4). Unreferenced issues were not reproducible in latest version of software. Issues referencing JQF-Zest [66] were previously found and reported by the authors of JQF-Zest.

Program	Issue #	JQF-Zest	CONFETTI	CONFETTI_{tgt}
ant	A1 [66]	100 %	100%	100%
bcel	B1 [66]	100 %	0%	0%
bcel	B2 [66]	100 %	100%	100%
bcel	B3 [8]	0 %	40%	0%
bcel	B4	0 %	80%	0%
bcel	B5 [7]	0 %	100%	5%
bcel	B6 [9]	0 %	100%	20%
closure	C1 [66]	100%	100%	100%
closure	C2 [66]	90%	5%	85%
closure	C3 [66]	80%	45%	70%
closure	C4 [1, 2]	0%	95%	45%
closure	C5 [3]	0%	90%	15%
closure	C6	0%	5%	0%
closure	C7 [4]	0%	100%	20%
closure	C8 [17]	0%	100%	0%
closure	C9 [5]	15%	20%	15%
closure	C10	0%	100%	5%
closure	C11 [14–16]	0%	100%	0%
closure	C12 [13]	0%	35%	0%
closure	C13	0%	20%	0%
closure	C14,C15	0%	5%	0%
rhino	R(1-4) [66]	100%	100%	100%

to the fuzzer can pay off: even though many of the hints tried at each position of each input may be irrelevant for detecting a bug, some of them do. Given that the design of the fuzzer is to execute as many inputs as quickly as possible, a diversity of hints can lead to a greater diversity in coverage, and a diversity in bugs found.

We found that 6 of the 16 newly discovered bugs had already been found and patched in the most recent development version of the respective fuzzing targets — an encouraging sign that developers care about the kinds of bugs that Confetti can find. We reported the remaining 10 bugs to the developers, and at time of writing 4 bugs in Closure have been fixed by developers and 3 have been acknowledged, 3 bugs in BCEL are awaiting acknowledgment. The Closure developers found the bugs discovered by Confetti to be quite interesting, and in their investigation of Issue C11, found a separate (but related) bug that they have already fixed [18]. This is a testament to Confetti's ability to find truly unexpected behaviors, thereby revealing latent software errors, and contributing to the betterment of software quality. We describe several of the newly found bugs here to provide some more intuition into Confetti's performance.

In many cases, Confetti found these bugs thanks to taint tracking, finding special strings like `arguments`, `jscomp.reflectProperty` and `goog.reflect.objectProperty`. It is likely that these strings would trigger these bugs in both the Confetti and Confetti$_{tgt}$ runs. This is shown in the results in Table 2, in which there are a subset of bugs that Confetti and Confetti$_{tgt}$ do find with similar frequency that JQF-Zest was not able to find (Issues C4, C5, C8).

Bugs C4 and C5 are interesting, but technically could have also been detected at the same frequency if the strings `arguments` and `goog.reflect.objectProperty` were in the fuzzer's dictionary. Issue C7 presents an example that could not be detected with a dictionary. `$jscomp$` is used as an internal constant that Closure Compiler uses to construct internal aliases for arguments to functions. By supplying both `inner$jscomp$1` and `inner` as arguments to a function, the compiler throws an exception because it tries to construct a map of argument names and if `inner$jscomp$1` is supplied as the first argument, it will fail to insert the second argument name, leading to a `RuntimeException`. Note that, in this case, no dictionary-based approach could detect this bug, as the bug is only triggered if two arguments are specified, with the first argument matching the second argument, plus the suffix `$jscomp$`.

Several bugs were detected only by Confetti's global hinting strategy (C6, C8, C11-C15). For example, consider Issue C11, which Confetti was able to find in 100% of runs, while Confetti$_{tgt}$ and JQF-Zest were unable to find it. A simplified input exercising this bug is `(((goog$dom$TagName$$_88a).length) += (this)))`. The string `goog$dom$TagName$$_88a` was extracted via taint tracking and added to the global dictionary. Later in the fuzzing run, the generator decided to use it as the left-hand expression of an addition assignment operator. During an optimization pass, the compiler is unable to satisfy the precondition that `(((goog$dom$TagName$$_88a) .length)` matches the type of `this` and throws an exception. Exercising this bug would not be possible without the decoupling of string hints to their respective parametric byte input positions. In Closure, this proves to be very successful in finding new bugs.

In BCEL, global hints led to the discovery of two bugs that only Confetti was able to find (Issues B3 and B4). Of the bugs that only Confetti and Confetti$_{tgt}$ found (B5 and B6), Confetti was able to find them with 100% repeatability across the 20 experimental runs. This suggests that global hinting is a powerful technique for revealing bugs with a high rate of repeatability within BCEL.

4.3 RQ3: Efficacy of Hint Strategies

While Confetti runs, it also collects basic statistics on the inputs generated: which strategies were used when generating each input, and which inputs were saved to the fuzzing population. Recall that JQF-Zest, like many other greybox fuzzers, saves an input to its population for later fuzzing if the input reveals new branch coverage, or if it increases the hit count of a previously covered branch by an order of magnitude. Since inputs are derived from existing inputs, it's possible that a single input has benefited from multiple hints, and multiple kinds of hints.

The left side of Table 3 shows the total number of inputs generated (across all 20 runs), along with the success rate for the targeted hint strategies (SMT, Char, String), for the global hint strategy, and overall, for random mutation. Some observations from this portion of the table are that SMT and Char mutations are relatively effective, that is, they are several orders of magnitude greater in their success. However, despite this, these strategies are rarely employed compared to the other mutation strategies due to SMT solving being expensive and/or finding certain paths to be unsatisfiable, or in the case of Char hints, simply being encountered in fewer places than string comparisons. The other mutation strategies — String, Global and Random, generate several orders of magnitude more inputs, as they leverage the throughput of the underlying greybox fuzzing framework upon which Confetti is built. String mutation strategies are particularly effective in Google Closure, and Global mutation strategies are an order of magnitude more successful in Closure than in any other target application.

However, simply considering the success rate of each hint strategy does not adequately capture its overall efficacy. For example, if coverage is quickly saturated during the fuzzing run (as we found in the case of Maven), *no* mutation strategy will be successful, since there is no new coverage to find. Success rates can also be misleading because they do not capture how frequently a kind of hint is available to be tried (again, particularly notable for SMT inputs), nor how often a hint is inherited by multiple derived inputs.

The right side of Table 3 presents an analysis of each of the inputs that were saved by Confetti. This metric captures both how often a hint is available, and also how often a hint is inherited by a child input. Note that since a single input might have multiple hints (and multiple kinds of hints), the sum of the number of saved inputs with each form of hint may be greater than the total number of saved inputs (or fewer, in the case of Rhino, where some saved inputs had no hints). Of the targeted hint strategies, we can see that while SMT and Char targeted hints had the highest success rates, they are represented by only a relatively small proportion of the saved inputs. Since the fuzzer can generate and test inputs extremely quickly, running up to several thousand inputs per second, it's possible that inputs that *could* have been generated by the SMT solver were instead, first generated by chance, perhaps thanks to a different hint. There is an interesting exception to this in the case

Table 3: Hint information for each new saved (coverage-revealing or hit-count increasing) input, aggregated across 20 runs. The left side shows the success rate of each mutation strategy in creating inputs that reveal new coverage. Each saved input might have multiple hints; the right side reports on the number of saved inputs with each kind of hint (including inherited hints).

Program	Total # Inputs Generated	Success Rate of Mutation Strategy					Total # Inputs Saved	With Targeted Hints			With Global Hints
		SMT	*Char*	*String*	*Global*	*Random*		*SMT*	*Char*	*String*	
ant	948,781,594	0.56%	5.32%	0.0099%	0.0011%	0.0013%	12,808	36	354	6,713	8,703
bcel	9,756,905,877	4.92%	7.47%	0.0107%	0.0003%	0.0003%	32,807	796	139	825	31,885
closure	249,762,647	5.61%	4.99%	0.1482%	0.0565%	0.0492%	144,857	828	2,506	43,167	117,601
maven	6,572,328,873	0.85%	3.22%	0.0015%	0.0004%	0.0004%	26,397	155	722	10,689	18,358
rhino	1,838,661,632	1.03%	10.07%	0.0093%	0.0017%	0.0019%	38,119	480	2,112	5,072	24,375

of BCEL, as the number of saved inputs for SMT targeted hints and String targeted hints are the same magnitude. BCEL is unique among the applications that we studied in that its input (Java class files) is a format that contains both strings *and* binary data. SMT-targeted hints particularly excel at covering new branches that rely on specific "magic" bytes, as opposed to strings.

Perhaps the most interesting takeaway from these statistics is the enormous proportion of saved inputs that contain global hints. This is encouraging evidence that supports our hypothesis that global hints are a useful form of guidance for fuzzers. However, simply because an input was saved with a global hint doesn't mean that this input needed that hint in order to produce the same coverage (and be saved) — it is possible that the hint is coincidental to the coverage, and that another string could have also resulted in the same coverage. We investigate this idea in greater depth in RQ4.

4.4 RQ4: Analysis of Inputs with Global Hints

RQ1 and RQ2 show that using global hints covers more branches and finds more bugs. RQ3 shows the kinds of hints in inputs that are saved (*i.e.,* are coverage-revealing), showing that most of those saved inputs include global hints. However, it is hard to conclude that the global hints are necessary: maybe a *randomly* generated input could have revealed that same coverage — how do we know that the global hint was relevant to revealing this coverage? With RQ4, we RQ4 examine this concern directly, studying the likelihood that a random fuzzer (that also benefits from targeted hints) could generate an input that reveals the same coverage as the input that CONFETTI generated using global hints.

For each input I across all of CONFETTI's runs that revealed new coverage *and* had global hints, we took the parent input P (that was mutated by CONFETTI into I) and fuzzed it 1,000 times using both random generation and targeted hints, observing the coverage of those inputs. 1,000 iterations is an order of magnitude more fuzzing iterations than Zest would apply in a single cycle, which we believe provides a reasonable upper-bound of the likelihood of the fuzzer without global hints generating an input that revealed that same coverage. If the fuzzed input never produces the same coverage, then we may have some confidence in the hypothesis that, for that input, global hints were necessary to achieve the same coverage. This allows us to distinguish between global hints that are clearly unnecessary and those that might have been useful for revealing new behaviors during the fuzzing campaign.

Table 4 shows the number and percentage of saved inputs that have global hints that could be replicated without those global hints trivially (on the first try), eventually within the 1,000 runs, and never within the 1,000 runs. On Ant, BCEL, Maven and Rhino, the majority of saved inputs can be replicated without global hints, trivially. This indicates that, in these applications, for most of the global-hint-containing inputs that revealed new coverage, the global hint(s) were definitely not necessary to produce that same coverage. However, we note that the surviving inputs in the "Never" column are still roughly comparable to the number of targeted hints shown in Table 3. This is perhaps evidence that global hints are *at least* as effective of a strategy as targeted hints in revealing new coverage.

Closure is the one exception to this trend, in which over 98% of saved inputs cannot be replicated without global hints. This is likely due to the high rate of implicit flows within Closure itself. A common pattern that we found in Closure is that all occurrences of the same identifier name in an input are mapped to the same object inside of the compiler — losing the precise mappings from each occurrence of that identifier in the input. Many of the bugs that only CONFETTI was able to find in Closure have similar properties. Our overall conclusion from this analysis is that CONFETTI's novel combination of both global and targeted hints is more effective than using only targeted or only global hints.

4.5 Data Availability

Our artifact contains the source code and dependencies for CONFETTI, our scripts to run experiments, our modifications to JQF and JaCoCo, and all data produced by our experiments [52]. To encourage re-use, CONFETTI is released under the BSD 2-clause license and our GitHub repository has a continuous integration workflow to run performance evaluations of pull requests [51].

Table 4: Post-mortem analysis of all saved inputs with global hints. For each input, we remove all global hints and attempt to replicate the same coverage by replacing the global hints with random strings from the fuzzer's dictionary. We show the number of those inputs with coverage replicable trivially (on the first try), eventually (within 1,000 tries), and never within those 1,000 tries.

Program	Saved Inputs Replicated Without Global Hints		
	Trivially	*Eventually*	*Never*
ant	6,664 (76.57%)	125 (1.44%)	1,914 (21.99%)
bcel	16,714 (52.42%)	1,440 (4.52%)	13,731 (43.06%)
closure	839 (0.71%)	668 (0.57%)	116,094 (98.72%)
maven	14,229 (77.51%)	173 (0.94%)	3,956 (21.55%)
rhino	19,980 (81.97%)	454 (1.86%)	3,941 (16.17%)

5 Discussion and Threats to Validity

Reliably evaluating fuzzers is difficult, since the process is non-deterministic. We mitigated this risk by following best practices: we ran our experiment 20 times, and reported in Table 1 only bugs found at least once in those 20 runs [50]. Confetti might have different performance on other programs: we used a benchmark of fuzzing targets used by prior work [66]. Our tools and data are available for others to replicate and expand on.

While our approach should be language-agnostic, we have only implemented it targeting programs that run in the JVM. We believe that the Confetti's approach could even be used for programs written in C, as shown by recent source code instrumentation-based approaches to concolic execution [68]. While we are hopeful that global hinting will be as significant of a hint strategy for other fuzzers (like AFL or libFuzzer), it is possible that there is hidden coupling between the success of global hinting and the design of the particular fuzzer that we extended (JQF-Zest). Like most other fuzzers, it will be difficult to apply Confetti to stateful applications in which separate inputs are related. Pairing Confetti with a checkpoint-rollback system [29] could ensure high-fidelity fuzzing.

It is interesting to consider why prior greybox fuzzers that leveraged taint tracking or concolic execution used it only for targeted guidance. One hypothesis is that, in other languages, it is difficult to identify bytes that are used to represent strings, versus binary data. However, popular fuzzers AFL and libFuzzer both already leverage statically-derived dictionaries [20, 54]. Hence, perhaps it is more likely that our approach of global hinting simply hasn't been tried yet, due to the concern that the global hint set would grow to such a large size to become unmanageable. Our experimental results seem to support the idea that including more strings in the global hint set (including those that may not be useful) is *more* beneficial than only considering targeted hints. Even if our results do not generalize to other languages, we note that Confetti is the only concolic-guided JVM-based fuzzer, and hence our findings still have a significant impact for any software engineers or researchers interested in fuzzing JVM-based code.

We did not carefully explore the configuration space for Confetti, and it is possible that its performance could increase or decrease on some or all fuzzing targets based on tunable parameters, such as the frequency at which hints are applied. We believe that this could be interesting future work, but feel that such an evaluation is outside of the scope of this paper.

Like JQF-Zest, Confetti assumes the availability of generators to exercise the programs under test. We do not see this as a significant limitation, however, due to the popularity of generator-based testing tools like JQF [65], ScalaCheck [60] and JUnit-Quickcheck [46]. Furthermore, our evaluation used only the pre-existing generators that were used in the original evaluation of JQF-Zest [66]. One hypothesis for the success of global hinting in our experiments is that the pre-existing generators were overly restrictive in the values that they could generate, and global hints provided a means to generate more diverse inputs. In this light, Confetti might be seen more as an approach to automatically improve the quality of existing generators, bypassing these restrictions. While we have not yet been able to design an experiment to confirm this result, we now believe quite strongly that future research in fuzzing JVM-based applications should focus on either approaches to evaluate and improve developers' existing generators and property tests, or to design new approaches that do not rely on those generators.

While we have very carefully tested our prototype implementation of Confetti, it is possible that our evaluation is affected by bugs that remain in Confetti or any of the other systems that we used (including JQF-Zest, JaCoCo and Phosphor). We analyzed the fuzzing results of both Confetti and JQF-Zest quite carefully, conducting thousands of short debugging runs, using JaCoCo to analyze the coverage (or lack thereof) of particular branches. We note that in addition to our own implementation bugs that we found and patched in Confetti, we also found several bugs in JQF-Zest and JaCoCo. For example: we found that JQF-Zest's coverage implementation did not correctly distinguish between the multiple cases of a single switch statement, and was generally prone to frequent collisions, where multiple branches used the same coverage counter. These issues resulted in JQF-Zest discarding many coverage-revealing inputs, rather than saving them and mutating them further. We implemented a new branch coverage instrumentation and runtime for JQF-Zest that eliminated these collisions, while also improving the fuzzer's execution speed by 7-10x. We used this enhanced coverage in our evaluation of JQF-Zest, and submitted this change as a pull request to the JQF-Zest maintainers, who are excited to merge it in to their next release [25].

As part of this JQF-Zest debugging, we also found a bug in JaCoCo that could cause branches to appear uncovered if the first statement enclosed by the branch was a method invocation which threw an exception. Tracing precise branch coverage in the presence of exceptional flow requires placing probes before and after any instruction that might throw an exception, which can decrease performance, and hence, this may not be desirable as a general feature. After considerable reflection, we determined that this behavior was acceptable within the scope of what JaCoCo aims to detect, and did not submit these changes to the JaCoCo developers. However, our branch of JaCoCo with the patch for this issue is in our artifact [52].

To help support future research and development of Confetti, JQF-Zest, and related fuzzers, we have created and shared a GitHub Actions workflow that automatically executes the entire evaluation that is described in this paper. We have found this workflow to be extremely useful in our development. For example: when debugging a memory leak in Confetti, we could easily launch many parallel experiments, collect metrics, and compare performance across branches. This workflow was also quite useful in our development of the faster, collision-free coverage implementation for JQF-Zest, which allowed us to easily create and share performance evaluation results in our pull request [25].

6 Related Work

In classic dynamic symbolic execution, for instance, as proposed by KLEE [32] or JPF-SE [22], programs are executed symbolically, by a special-purpose interpreter. Concolic execution, executes a program concretely, but uses runtime support to collect path constraints as they relate to the input, then later negates some of these constraints, solves them using an SMT solver, and executes the newly generated input [31, 71]. Hybridizing concolic execution and random testing/fuzzing was first proposed by Majumdar and Sen [57]. This work showed that once fuzzing saturates code coverage, concolic testing can help to discover new program states that random testing didn't otherwise find. More recent work has also explored hybridizing concolic execution with fuzzing [61, 62, 74], but CONFETTI is the first work to use the *global hinting* strategy to integrate concolic guidance with fuzzing.

Fuzzing parsers for well structured, human readable, input is challenging. One line of research aims to guide a fuzzers with a grammar that describes the input structure [41, 43, 67, 77, 78]. For instance, SKYFIRE used grammars to generate well formed inputs as seeds for AFL [77], and SUPERION integrated the grammar with AFL [78]. Such input grammars are required to be context-free, which limits their applicability. To address this limitation, previous work focused on learning tree models [67] or probabilistic context-sensitive grammars [43] from a corpus of valid seeds. In contrast, CONFETTI's generators are small programs that can generate sophisticated inputs (*e.g.,* any valid Javascript program) without the restrictions of context-free grammars.

Many other systems also combine symbolic or concolic execution with fuzzing [33, 35, 63, 72, 79, 80]. Perhaps most similar to CONFETTI is ANGORA, which, like CONFETTI, also uses taint tracking to collect path constraints [33]. CONFETTI differs from these prior systems in that it records strings generated by concolic execution as *global hints*, allowing these magic values to be used in elsewhere in the same or other inputs. In our evaluation, we found that this strategy accounted for most of the coverage-revealing inputs found by our fuzzer. These global hints are effectively a *dynamically generated* fuzzing dictionary — fuzzers like AFL [81], libFuzzer [55] and JQF-Zest [66] all allow developers to specify a pre-defined dictionary of strings that might be interesting to use in fuzzing. In our evaluation, all fuzzers were seeded with dictionaries by JQF-Zest's original authors, providing a realistic representation of the dictionaries that a developer would create.

Similarly, CONFETTI is not the first approach to combine taint tracking with fuzzing. Like CONFETTI, VUZZER combines taint tracking with fuzzing in order to target the fuzzer and determine magic bytes [69]. BUZZFUZZ uses taint tracking to identify which input bytes that flow into targeted branches, and then modifies those bytes directly [40]. Similarly, TAINTSCOPE uses taint tracking to identify inputs that flow through checksum-like routines and attempts to use a symbolic representation to ensure that the fuzzed inputs still pass those checksums [80]. Again, CONFETTI differs from all of this prior work in that it also introduces the notion of global *hints*, which repurpose values detected from taint tracking particular bytes of one input to be used when fuzzing other inputs. There have also been numerous advancements in fuzzer seed selection and scheduling, most of which are complementary to CONFETTI, and combinations of the approaches could be studied in future work. For instance, directed greybox fuzzing guides a traditional greybox fuzzer by casting guidance as an optimization problem, and hence does not require whitebox guidance at all [30].

CONFETTI ameliorates the implicit flow problem by loosening the coupling between values detected by taint tracking or concolic execution and the part of a particular input where that magic value should be applied. Mathis et al.'s LFUZZER addresses taint tag loss through implicit flows in input tokenization by automatically identifying routines that parse input characters into tokens and propagating taint tags along those conversions [58]. Like CONFETTI, LFUZZER also adds these tokens to a global dictionary to use in fuzzing. In our evaluation, we found that CONFETTI's hints revealed bugs in program logic *after* tokenization and parsing, for instance, in the optimization phase of the Closure compiler — outside of the tokenization routines that LFUZZER would target. Other taint-tracking-based fuzzers attempt to address the implicit flow problem by inferring control dependencies between branches and input bytes, by comparing coverage results while mutating inputs [34, 39]. However, these systems can only detect that relationship *after* the fuzzer succeeds in covering the branch. We demonstrated that CONFETTI's global hints can be used to reveal branches that the fuzzer could not otherwise. Future work might combine CONFETTI with heuristics for selectively propagating taint tags through implicit flows [24, 49].

While popular fuzzers like AFL target x86 binaries, there remains a need for fuzzers targeting higher level languages like Java. Java PathFinder (JPF) [76] is a model checker for Java programs that uses a custom-built interpreter to collect and solve path constraints in order to explore different program states. JPF has been a significant resource for the Java testing community, and has been extended in many ways to support various forms of dynamic symbolic execution [22, 48, 56]. Prior concolic execution tools for Java like JCUTE [70], CATG [73] and CINGER [21] used instrumentation-based approaches to track constraints in a limited subset of classes. In contrast, CONFETTI uses a dynamic taint tracking system to track path constraints, and does so in all classes. To our best of our knowledge, CONFETTI is the first system that supports concolic execution of real-world Java programs like those in our evaluation.

7 Conclusion

CONFETTI is a concolic-guided fuzzer for JVM software that generates inputs covering more branches and revealing more bugs than the existing state-of-the-art JVM fuzzer. Through our empirical studies, we have identified that CONFETTI's novel *global hinting* mechanism yields a significant improvement in coverage and bug finding compared to the state-of-the-art approach of targeted hinting. Although we have only explored global hinting in the context of a single fuzzer (JQF-Zest) and a single language (Java), we believe that there is strong evidence that this approach will be quite successful in other fuzzing domains, too. Based on our analysis of the failures that could be detected only by CONFETTI (and not by the variant without global hints), we have a strong intuition that the same kinds of programming patterns that restrict the efficacy of targeted hints in our experiments occur in other applications and languages, as well. We hope that our open-source release of CONFETTI and its CI workflow will help to support the growing community of practitioners and researchers engaged in fuzzing JVM-based software [51, 52].

Acknowledgments

We would like to thank Katherine Hough for valuable feedback on the design, implementation and debugging of CONFETTI, and Rohan Padhye for assistance merging our fixes into JQF. This work was funded in part by NSF CCF-2100037 and NSF CNS-2100015.

References

[1] 2019. Google Closure Issue Tracker Issue Number 3375. https://github.com/google/closure-compiler/issues/3375.
[2] 2019. Google Closure Issue Tracker Issue Number 3380. https://github.com/google/closure-compiler/issues/3380.
[3] 2019. Google Closure Issue Tracker Issue Number 3455. https://github.com/google/closure-compiler/issues/3455.
[4] 2020. Google Closure Issue Tracker Issue Number 3591. https://github.com/google/closure-compiler/issues/3591.
[5] 2020. Google Closure Issue Tracker Issue Number 3592. https://github.com/google/closure-compiler/issues/3592.
[6] 2021. Apache Ant. https://ant.apache.org/.
[7] 2021. Apache BCEL Issue Tracker Issue Number 357. https://issues.apache.org/jira/projects/BCEL/issues/BCEL-357.
[8] 2021. Apache BCEL Issue Tracker Issue Number 358. https://issues.apache.org/jira/projects/BCEL/issues/BCEL-358.
[9] 2021. Apache BCEL Issue Tracker Issue Number 359. https://issues.apache.org/jira/projects/BCEL/issues/BCEL-359.
[10] 2021. Apache Commons Byte Code Engineering Library. http://commons.apache.org/proper/commons-bcel/.
[11] 2021. Apache Maven. http://maven.apache.org/.
[12] 2021. Google Closure. https://developers.google.com/closure/compiler.
[13] 2021. Google Closure Issue Tracker Issue Number 3857. https://github.com/google/closure-compiler/issues/3857.
[14] 2021. Google Closure Issue Tracker Issue Number 3858. https://github.com/google/closure-compiler/issues/3858.
[15] 2021. Google Closure Issue Tracker Issue Number 3859. https://github.com/google/closure-compiler/issues/3859.
[16] 2021. Google Closure Issue Tracker Issue Number 3860. https://github.com/google/closure-compiler/issues/3860.
[17] 2021. Google Closure Issue Tracker Issue Number 3861. https://github.com/google/closure-compiler/issues/3861.
[18] 2021. Google Closure Issue Tracker Issue Number 3862. https://github.com/google/closure-compiler/issues/3862.
[19] 2021. Mozilla Rhino. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.
[20] AFL Contributors. 2021. AFL Dictionaries. https://github.com/mirrorer/afl/blob/master/dictionaries/README.dictionaries.
[21] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666
[22] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF-SE: A Symbolic Execution Extension to Java PathFinder *(TACAS'07)*. Springer-Verlag, Berlin, Heidelberg, 134–138.
[23] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. 2021. JavaSMT 3: Interacting with SMT Solvers in Java. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 195–208.
[24] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Strict Control Dependence and Its Effect on Dynamic Information Flow Analyses *(ISSTA '10)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1831708.1831711
[25] Jonathan Bell. 2022. Faster, collision-free coverage instrumentation, Pull Request #171 on rohanpadhye/JQF. https://github.com/rohanpadhye/JQF/pull/171.
[26] Jonathan Bell. 2022. Increase maximum depth of XML files generated for Maven, Pull Request #165 on rohanpadhye/JQF. https://github.com/rohanpadhye/JQF/pull/165.
[27] Jonathan Bell and Gail Kaiser. 2014. Phosphor - GitHub. https://github.com/gmu-swe/phosphor.
[28] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs *(OOPSLA '14)*. ACM, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212
[29] Jonathan Bell and Luís Pina. 2018. CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs *(ECOOP '18)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
[30] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing *(CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020
[31] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production *(ICSE '13)*. IEEE Press, 122–131.
[32] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756
[33] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.
[34] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing Deeply Nested Branches *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 499–513. https://doi.org/10.1145/3319535.3363225
[35] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code *(ICSE '19)*. IEEE Press, 736–747. https://doi.org/10.1109/ICSE.2019.00082
[36] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[37] Stephen Dolan. 2017. Property fuzzing for OCaml. https://github.com/stedolan/crowbar.
[38] Andrea Fioraldi, Dominik Maier, Heiko Eibfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.
[39] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan
[40] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546
[41] Patrice Godefroid, Adam Kiezun, and Michael Levin. 2008. Grammar-based Whitebox Fuzzing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 43, 206–215. https://doi.org/10.1145/1379022.1375607
[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036
[43] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing *(ASE 2017)*. IEEE Press, 50–59.
[44] Google. 2022. honggfuzz. https://honggfuzz.dev.
[45] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 49–64. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller
[46] Paul R. Holser. 2010. junit-quickcheck: Property-based testing, JUnit-style. https://github.com/pholser/junit-quickcheck.
[47] Katherine Hough and Jonathan Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 26 (dec 2021), 43 pages. https://doi.org/10.1145/3485464
[48] Karthick Jayaraman, David Harvison, and Vijay Ganesh. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java. In *Proceedings of the 1st NASA Formal Methods Symposium (NFM)*.
[49] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*.
[50] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804
[51] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2019. CONFETTI: Amplifying Concolic Guidance for Fuzzers. https://github.com/neu-se/CONFETTI.
[52] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. https://doi.org/10.6084/m9.figshare.16563776
[53] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 181 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360607
[54] libFuzzer Contributors. 2021. libFuzzer Tutorial. https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md#dictionaries.
[55] LLVM Project. 2019. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[56] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 442–459. https://doi.org/10.1007/978-3-662-49674-9_26
[57] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 416–426. https://doi.org/10.1109/ICSE.2007.41
[58] Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning Input Tokens for Effective Fuzzing *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 2737. https://doi.org/10.1145/3395363.3397348
[59] Mountainminds GmbH & Co. KG and Contributors. 2021. JaCoCo Java Code Coverage Library. http://www.eclemma.org/jacoco/.

[60] Rickard Nilsson. 2019. ScalaCheck: Property-based testing for Scala. https://www.scalacheck.org.

[61] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: Complexity Analysis with Fuzzing and Symbolic Execution *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 322–332. https://doi.org/10.1145/3213846.3213868

[62] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid Differential Software Analysis *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1273–1285. https://doi.org/10.1145/3377811.3380363

[63] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1475–1482. https://doi.org/10.1145/3167132.3167289

[64] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF + Zest: Coverage-guided semantic fuzzing for Java. https://github.com/rohanpadhye/JQF.

[65] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[66] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[67] Jibesh Patra and Michael Pradel. 2016. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

[68] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[69] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf

[70] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*, Thomas Ball and Robert B. Jones (Eds.). 419–423.

[71] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C *(ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[72] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[73] H. Tanno, X. Zhang, T. Hoshino, and K. Sen. 2015. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 717–720.

[74] Willem Visser and Jaco Geldenhuys. 2020. COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 373–377.

[75] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

[76] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engg.* 10, 2 (April 2003), 203–232. https://doi.org/10.1023/A:1022920129859

[77] J. Wang, B. Chen, L. Wei, and Y. Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

[78] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[79] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 61–64. https://doi.org/10.1145/3183440.3183494

[80] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2011. Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution. *ACM Trans. Inf. Syst. Secur.* 14, 2, Article Article 15 (Sept. 2011), 28 pages. https://doi.org/10.1145/2019599.2019600

[81] Michal Zalewski. 2019. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[82] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2019.23504