June: A Type Testability Transformation for Improved ATG Performance

Dan Bruce
Microsoft
University College London
UK
dan.bruce@microsoft.com

David Kelly
King's College London
UK
david.a.kelly@kcl.ac.uk

Hector Menendez
King's College London
UK
hector.menendez@kcl.ac.uk

Earl T. Barr
University College London
UK
e.barr@ucl.ac.uk

David Clark
University College London
UK
david.clark@ucl.ac.uk

ABSTRACT

Strings are universal containers: they are flexible to use, abundant in code, and difficult to test. *String-controlled programs* are programs that make branching decisions based on string input. Automatically generating valid test inputs for these programs considering only character sequences rather than any underlying string-encoded structures, can be prohibitively expensive.

We present June, a tool that enables Java developers to expose any present latent string structure to test generation tools. June is an annotation-driven testability transformation and an extensible library, JuneLib, of structured string definitions. The core JuneLib definitions are empirically derived and provide templates for all structured strings in our test set.

June takes lightly annotated source code and injects code that permits an automated test generator (ATG) to focus on the creation of mutable substrings inside a structured string. Using June costs the developer little, with an average of 2.1 annotations per string-controlled class. June uses standard Java build tools and therefore deploys seamlessly within a Java project.

By feeding string structure information to an ATG tool, June dramatically reduces wasted effort; branches are effortlessly covered that would otherwise be extremely difficult, or impossible, to cover. This waste reduction both increases and speeds coverage. EvoSuite, for example, achieves the same coverage on June-ed classes in 1 minute, on average, as it does in 9 minutes on the un-June-ed class. These gains increase over time. On our corpus, June-ing a program compresses 24 hours of execution time into *ca.* 2 hours. We show that many ATG tools can reuse the same June-ed code: a few June annotations, a one-off cost, benefit many different testing regimes.

CCS CONCEPTS

• **Software and its engineering → General programming languages**; **Empirical software validation**; *Search-based software engineering*.

KEYWORDS

June, Testing, Search-based Testing, Java, Automated Test Generator, Grammars, Strings

ACM Reference Format:
Dan Bruce, David Kelly, Hector Menendez, Earl T. Barr, and David Clark. 2023. June: A Type Testability Transformation for Improved ATG Performance. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3597926.3598055

Dan Bruce, David Kelly, Hector Menendez, Earl T. Barr, and David Clark

1 INTRODUCTION

Testing dominates software assurance. Testing is only as good as its test suite, which is costly to write and maintain. To alleviate this cost and improve testing, Automatic Test Generation (ATG) aims to create test suites with minimal user assistance. ATG tools, like EvoSuite [20], utilise criteria such as code coverage to guide test generation [4]. ATG has improved testing practice, but progress in ATG has run up against the problem of *structured input generation (SIG)*: the difficulty of constructing objects that satisfy program guards. Structured inputs range from IP addresses, most result sets of a database query, to JSON. Developers often store these structures in strings. The tool sees only a string, whereas there is, in fact, a predictable structure: these are *latently structured strings.*

Recent work in ATG has focused on high-level input grammars [22, 45]. This high-level approach is much used in fuzzing. Fuzzers, and techniques build on top of fuzzers, such as *Zest* [36], target program access points with strings generated through a grammar, mutating them to penetrate deeper into the control flow graph. This approach works well in *system* testing, whereas the tools of interest to us do *unit* testing. In this paper, we argue that for unit testing, this *top down* approach, just like *trickle down* economics does not adequately allocate resources. Using much smaller,

Dan Bruce and David Kelly are joint first authors.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598055

local, grammars provided directly over method parameters, we emphasise the *middle out* or *bottom up* approach. Our results suggest that bottom up reasoning allows much finer grained control when testing at the method level (Section 5).

To utilise local grammars, we present June, *Java string tuning*. June is a testability transformation [25] and support library, JuneLib, written in Java. June finds suitable annotation locations and handles the code transformation for the developer. June captures string structure in lightweight, *in-language* annotations and exposes that structure to an ATG tool, such as EvoSuite. These local grammars are deliberately simple, expressing concepts such as *number* or *delimited* string. They are not antagonistic with top down grammar generation, but provide useful guidance deep inside the program. A part of the testing resource, otherwise spent uselessly generating strings in some potentially large master grammar, is now spent in much simpler, more locally precise, templates.

Over a commonly used corpus of Java programs, SF110 (Section 4.2), approximately 6% of classes have string parameters that affect control flow. Many of these string parameters occur at I/O points; efficiency on these paths are important for safety and the scalability of continuous integration, or CI (Section 2). We show that June makes EvoSuite over 10× faster on June-ed classes, reaching a day's worth of coverage in under 2 hours 30 minutes. Over less generous testing times, June has a notable impact: the coverage EvoSuite achieves after 2 minutes on un-June-ed classes can be had after just 22 seconds.

June is not burdensome to use: in our corpus, we required an average of only 2 annotations per class, and the variable name frequently gave us the annotation directly. With this small effort, we find that EvoSuite can generate tests to the same degree of coverage for the corpus in just 13 minutes, rather than an hour. As we add annotations only once, in a continuous integration context this small cost is amortised over CI runs (see Section 2).

June is a testability transformation which refines type information, so it is not simply an EvoSuite extension. It works with different ATG tools using the same annotated source code. It is tool agnostic. This marks it as different from generators used for specific tools, such as the parametric generators introduced in Zest (Section 6). The generators in Zest are written in the tool itself, whereas June generators are independent of the testing tool. We evaluate this agnosticism by experimenting with 4 different ATG tools. June results in improvements for all. The generality of testability transformations means that there is nothing inherently *Java* about our approach. ATG tools for other languages should also benefit from string tuning for testing.

June consists of a Maven plugin that leverages Java's native annotation language (Section 3) and a library, JuneLib, of *SafeStrings* [28]. *SafeStrings* are an elegant, language agnostic way to capture latent structure and make it available to testing tools. By construction, JuneLib captures common structured strings (Section 3.1). To use June, a developer need only select or extend an existing annotation from JuneLib. Indeed, JuneLib already defined *SafeStrings* for every latently structured string in our set of Java classes (Section 3.1).

Our principal contributions are:

- We present June, the first bottom-up framework for exposing latent string structure to ATG tools, and JuneLib, a library of *SafeStrings* that captures all of the structured strings in our corpus;
- We show that June greatly increases the efficiency of EvoSuite and other ATG tools such as Randoop and JQF, in terms of time, work done, and overall coverage;
- We show that June has minimal user effort, with fewer than 10% of method parameters in string-controlled classes requiring annotations.

Listing 1: Example method signature with email address parameter. Taken from the `EmailAddressLocalServiceImpl` class with *SF110*$_{30}$.

```
protected void validate(
  ... String address, ...) {

  if (!Validator.isEmailAddress(address)) {
    throw new EmailAddressException();
  }
  /* method body continues */
```

2 MOTIVATING EXAMPLE

We motivate June with a common testing situation: continuous integration. A developer is responsible for a project, and wants to ensure that the code is well tested before every release. Unit tests cover some amount of the code base, regression tests cover some more, and some paths require manual testing. These manual paths are often the system critical pathways of a project, those expected to be most commonly used, or with the greatest density of "business logic". One can include ATG tools, such as EvoSuite, in the CI, to generate tests for classes on the critical pathways.

Another technique for including ATG tools in CI is to run them on any classes which have been changed as part of a pull request. These two methods are not mutually exclusive and can be combined so that tests are generated on every pull request for all critical classes (as defined by developers) and all classes changed as part of the pull request. This increases confidence in code, and helps catch bugs before the code hits production. Common practice suggests 2 minutes [21] as a reasonable time limit for ATG. Two minutes per class quickly accumulates, however, especially when run on every pull request to the main branch. Using June, one can reduce the time given to an ATG tool, yet achieve the same or higher coverage. We examine how to uses June on a class with email addresses encoded as a string.

Alternatively, in a software setting with low code churn, where an ATG tool runs only once to generate a test suite, June-ed code still allows a tool to create a higher coverage test suite within the same time frame.

Emails are all over the internet. One finds typical email handling code, like that in Listing 1, in the *SF110*$_{30}$ corpus of Java classes (Section 4.2). The code in Listing 1 uses an external library to validate the email address. ATG tools, such as EvoSuite [18], cannot harvest useful information from external library calls: they are not instrumented and usually return either a boolean or simply error.

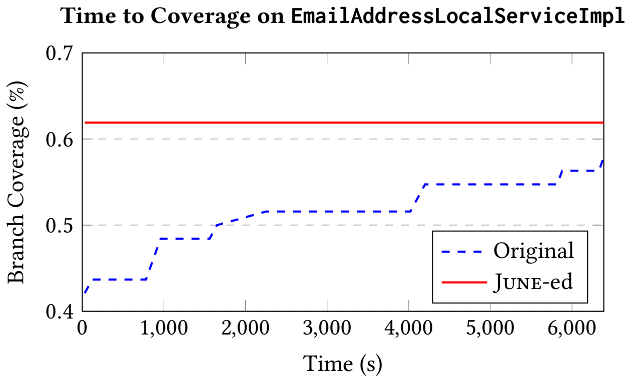

Figure 1: Coverage over time of original and June-ed versions of `EmailAddressLocalServiceImpl`. Once a class has been June-ed EvoSuite can easily generate structurally correct strings through the JuneLib constructors. EvoSuite hits maximum coverage of the June-ed version immediately.

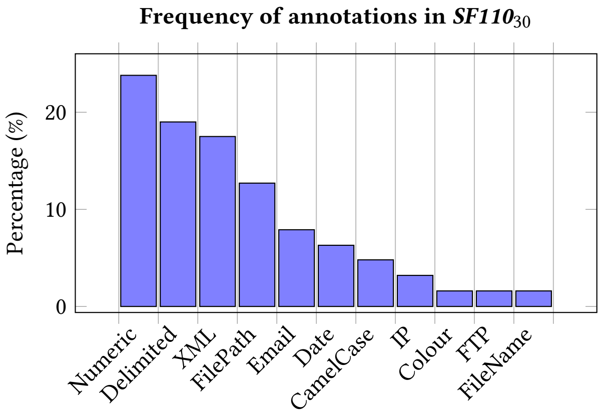

Figure 2: All 11 types used in *SF110*$_{30}$ and the percentage of annotations for which they account.

Booleans carry insufficient information to calculate the 'distance' an input is from being satisfactory.

An alternative is to use regular expressions [40]. Internal validation via regex occurs frequently, appearing in an estimated 30-40% of JavaScript and Python projects [13, 16]. Internal validation via regular expression is difficult to maintain. One possible regular expression matching an email address is in

```
/^[-a-z0-9~!%^&*_=+}{\?]+(\.[-a-z0-9~!$%^&*_=+}
↪  {\?]+)*@([a-z0-9_][-a-z0-9_]*(\.[-a-z0-9_]+[a-z]⌋
↪  [a-z])|([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9⌋
↪  ]{1,3}))(:[0-9]{1,5})?$/i]/
```

It is unlikely that a developer would be willing to use this in code, even in a simplified form. A regex as complicated as that above is difficult to understand, alter, or even read. Generating strings from a complex regex is also expensive and difficult (Section 6).

We found the email regular expression through online search. The ease with which we found it suggests that email validation occurs quite often in real code. In fact, we suspect that strings are not really a wild west of unstructured inputs, but that a few different types of structure appear again and again. One need only think of the prevalence of JSON or XML. We took a data-driven, physical sciences approach to examine this phenomenon. We conducted a study to find which string-encoded latent structures, and their frequency, we could discern in method parameters in SF110. This study informed the design of JuneLib (Section 3.1). JuneLib's design should obviate the need to write custom grammar generators for method parameters, as JuneLib's *SafeString*'s are composable.

When we applied JuneLib to *SF110*$_{30}$ we found the distribution in Figure 2. Remarkably, the distribution of structured string types in our corpus follows a power law. The conclusion is a *SafeString* support library can be small, and a developer need only apply a handful of its SafeStrings to get many benefits. With June, the developer's only requirement, falling short even of an obligation, is usually just to decide which *SafeString* to use and to annotate accordingly. It is not an obligation, as adding annotations, assists, whereas not providing them just falls back directly on the underlying tool. One does not even need to write a test harness, as one would for a fuzzer.

June helps with random test input generation by breaking a structured string down into mutable and immutable sections, leaving to random search the smaller problem of creating local, unstructured, substrings. The June constructor simply slots the mutable sections into the immutable framework of the June string. As the structure and immutable elements are present by construction, it vastly decreases the search space. This allows ATG tools to search over valid instances of a small grammar rather than search for the grammar itself.

Developers annotate each method parameter which they want an ATG tool to use a June type for when testing. The cognitive effort is low: developers have knowledge of the code and many latent grammars are obvious to the human eye. For example, it takes no effort to identify that the parameter 'address' in Listing 1 is an `@EmailString`. Even though the parameter is not named 'email', the first line calls a method `Validator.isValidEmail()` which is a clear indicator of the anticipated content of the input. This first line will block ATG tools generating tests which cover the rest of this method beyond the if block, until a generated input passes email validation. As described earlier, the criteria for a valid email is not simple, and the call to an external method provides little information to a testing tool. Using June will unlock the region of code beyond this input validation by ensuring valid emails are used as inputs.

Not only is identifying the correct *Java SafeString* to use low effort, but we find the same June types appear frequently. Figure 2 shows the distribution of June types used in our experiments Section 4. There were 64 method parameters annotated with 11 different June types, with the most popular 3 types accounting for over 50% of all annotations.

Helping ATG Tools. A compiled June-ed class contains a mixture of June objects and strings. These objects provide information to the ATG tool, allowing better testing efficiency. To build an object required as an input, ATG tools commonly call the object's constructor, or a static method which returns a new instance of the object. With carefully crafted constructors and static methods, JuneLib assists ATG tools to efficiently and effectively generate structured string inputs.

It is still possible for ATG tools to generate random strings that are invalid: for example, given an email address *Java SafeString*, a tool can generate a string with an "@" symbol for the name. This violates the structural requirements of an email address. June runs validation checks, ensuring that a type error is thrown, informing the testing tool that the input was invalid. Complex tools, such as EvoSuite, can harness this information to avoid generating similar invalid inputs. Even when there is a "clash" of this nature, the testing tool does not waste time testing the program under test with the invalid input. Instead a short feedback loop between June and the testing tool ensures a structurally valid input before executing the program under test. There is one caveat to this. Sometimes invalid inputs are needed to cover regions of code. Therefore each June type has one constructor which takes a string and does not modify it, in essence allowing any string to be used, regardless of whether it is a valid instance of the June type.

June-ed code is very similar to the original source. We use Java annotations [46] to provide information for the June transformation (Section 3). The transformed code provides everything a testing tool needs to generate correctly structured strings. June changes type signatures and inserts calls to constructors, while still guaranteeing interoperability with the string type via use of `toString()` methods. The transformation is only used for testing and has no effect on production code, ensuring there are no possible side affects.

The `EmailAddressLocalServiceImpl` class from `liferay`, found in *SF110*$_{30}$ (Section 4.2), contains methods with string parameters. On the class, EvoSuite [18], an industry standard ATG tool, achieves branch coverage of approximately 58% in four hours. Branch coverage in the first 30 seconds is approximately 42%. It takes a large allocation of resources to increase coverage of the class by 16 percentage points. This is understandable: by default, EvoSuite only "knows" that it is targeting a string, rather than an email string, and, as we see in Listing 1, any string that is not a valid email throws an exception. With access to the additional information provided by June, EvoSuite can correctly generate well-formed email addresses immediately, greatly reducing the cost of testing this method, freeing it to spend resources on other tasks (Section 5). It then achieves better coverage, 62%, not in 4 hours, but in the first 30 seconds (Figure 1). This reduced cost and increased coverage makes CI a viable option.

The developer provided annotations on three of the seven methods in the class. Each of these methods require only one annotation. In return for this marginal effort, using EvoSuite, coverage is greater in 30 seconds than in a full 24 hours of computation on the un-June-ed class.

3 JUNE: DESIGN AND DEPLOYABILITY

We designed JuneLib to capture as many common latently structured string types as we could. Making JuneLib comprehensive allows the developer to focus solely on the benefits of using it, without the necessity of looking inside the library internals. We discuss first the design of JuneLib, and then look at how to use June when testing.

3.1 JuneLib Design

JuneLib is a collection of *Java SafeString* [28] definitions: each *Java SafeString* is a class that inherits from the core `SafeString` class. The `SafeString` class has the same methods as the native Java `String` class, ensuring easy interoperability. JuneLib was constructed to cover as many commonly encountered latently structured strings as possible.

We did not initially know which *Java SafeString* types to include in JuneLib. In order to build a picture of the landscape, we chose 500 classes, uniformly at random, from the initial dependency filtering of SF110. We then manually examined each class to see what strings were used and to ascertain their structure. Due to the large number of classes to examine, we gave ourselves one minute to learn the structure of the string. We chose this small time budget to reflect what we believe to be a maximum realistic time allocation for developers. Frequently, identifing the string was an easy task, as the parameters usually had informative names (*i.e.* an XML string parameter was called xml). On other occasions, we had to understand the structure from looking at the control flow of the method. We implemented a *Java SafeString* for each latently structured string that we found in the 500 classes, a total of 28 unique *Java SafeStrings*. We only used 11 in the final experiments (Section 5.2). Figure 2 show the distribution of string types as finally used in our experimental subset.

We did not create a *Java SafeString* for every structured string that we encountered. An example of this is SQL. Java already has good mechanisms for handling SQL, and we did not want JuneLib to encourage the bad programming practice of storing executable code in string form. The target of *SafeStrings* is non-executable data stored in strings: our design of JuneLib respects this intention.

Evaluation. As we populated JuneLib based on what we found in SF110, we examined its generality by sampling 10 large, open source web portal projects uniformly at random from Github. This collected a more modern code collection that SF110. We chose web portal projects as these reflect the programs also found in SF110, and we did not wish to compare apples with oranges. Following static analysis, we found that the number of methods over the entire corpus that featured string dependencies according to June's static analysis was essentially the same as in *SF110*$_{30}$, approximately 6%. String type overlap between *SF110*$_{30}$ and the new classes was 90%. Interestingly, XML had been largely replaced by JSON in the more modern corpus, indictating that even strings are subject to changing fashions.

A developer might need a domain specific *Java SafeString* not in JuneLib. In our experiments (Section 4.3), we did not need to create any new *Java SafeStrings* after the initial creation of JuneLib. A *Java SafeString* is a combination of recogniser, structure and method for serialising the structure, such that the output of the serialiser is accepted by the recogniser. What this means in practice is that we have a type of string (*i.e.* email), a parser for that type, an object that stores the string's AST, and a special method, `cast()`, in Java a synonym for `toString()`, that recreates the original string from the object in a way acceptable to the parser.

3.2 June Design

At its heart, June consists of the core library of *Java SafeString* definitions, JuneLib (Section 3.1), and a Maven plugin to perform the code transformation and manage the build process. Figure 3 shows a typical workflow. Developer input is only required in one location, the process is otherwise automatic. To utilise the Maven plugin, the developer need only copy the XML configuration into the project pom.xml. This plugin performs a flow analysis to help the developer know *what* and *where* to annotate. As long as these types do not change (*e.g.* the developer stops using XML and rewrites the code to use JSON), they can remain in the code. They do not need to change for every test generation.

Dependency Analysis. We use Soot [43] to perform a flow dependency analysis on Java classes. June identifies classes containing at least one method that manipulates strings. By "manipulate", we mean that the method has at least one string parameter and one or more conditionals access that string input. Once June identifies string-controlled methods, the user must annotate them. Annotations use the native Java annotation language. For example, the user would change

```
String foo(String postcode) { ... }
```

to

```
String foo(@PostCodeSafeString String
    postcode) { ... }
```

The addition of annotations to source code is the only direct intervention to the source that the programmer needs to make. For the purpose of increasing ATG efficiency, we only annotate parameters. We show in Section 5.2 that minimal annotations are required, only 2.1 annotations on average per class in *SF110*$_{30}$.

June's Program Transformation. Once the annotations are in place, June transforms the program, using Spoon [37]. June is a source level program transformation to facilitate error handling. As June's source level output must be compiled, it allows compile time errors, such as type errors, to be caught. Further, June preserves control flow in the source code and has minimal footprint. Any errors generated are directly mappable back to the original source code. This helps the debugging process: the developer never needs to have direct contact with the June-ed code, unless that is desired.

June's transformation performs three principal tasks. First, it desugars annotations in type signatures to replace them with the appropriate types. Second, within a method, if a June variable needs to be a string, the `toString()` method is called. Third, where the method is called from elsewhere within the code, June checks whether the value provided as input is already of the correct June type. If it is not, June passes the string value into the appropriate JuneLib constructor, and uses this new variable as input parameter for the annotated method. The first part of the transformation is what provides an ATG with additional information. The second and third parts allow interoperability between methods within a class which have been annotated.

There is a alternative method for performing the first step. This selects a constructor from the JuneLib type, and then replaces the string parameter with all of the parameters that constructor. Then, June adds a new line to the start of the method to create the *Java SafeString* typed variable. This option is configurable and depends on the tool. The second and third step can then continue as previously explained. The benefit of this method is that tools which cannot generate objects, such as Randoop, can still reap the rewards of June annotations. We essentially embed the constructor for the June type in the method under test using only primitive types. To write a new *Java SafeString*, no understanding of the ATG tool being used is required. A developer need only understand the latent structure sufficiently well that they can write a constructor which builds valid instances. Selecting inputs which form the mutable parts of the grammar reduces the search space and the constructor combines these with the immutable parts to form a valid instance – both are independent of tool/approach.

Customising June. Allowing new types to be built via the annotation language provides great flexibility to June. It is easy, for example, to create a new subtype of delimited string by specifying the delimiter. Moreover, one can specify the minimum and maximum number of fields. The type of the field in a delimited string is also specifiable. At the moment, this is limited to one type, *i.e.* each field must be of the same type, but we hope to relax this constraint in future work. One could use the *Java SafeString* annotation

```
@DelimitedSafeString(min=4,max=4,delim=",")
```

to model a string that must have four comma separated fields. This ability is powerful, if the previous annotation was ammended so that all fields must be numeric the type becomes very close to an IPv4 address. However, `IPv4SafeString` has additional constraints on the fields ensuring the values are within the allowed range.

If a developer has needs which cannot be fulfilled by the existing annotations they are able to create new *Java SafeStrings*. To do so is relatively easy, they create a new class which extends the SafeString base class and ensure that constructors, validators and the `toString()` methods are all implemented. It is however important to consider how ATG tools will interact with the new type, and therefore consider how to reduce the searchspace as much as possible.

4 EXPERIMENTAL SETUP

We choose four ATG tools (Section 4.1) and the SF110 corpus (Section 4.2) to investigate the efficacy of June and the tool agnosticism of JuneLib. Our choices were guided by a few key requirements: the fundamental precondition of our technique is that the ATG tool uses methods as the entry points to generate unit tests, and is able to generate inputs based on the method signature. This is because June converts Java strings into specialised `Objects`, or the required primitive inputs to create a *Java SafeString*. Therefore, in order for an ATG tool to benefit from a June-ed program, it must be able to generate method sequences and provide varying primitive and string parameters. This allows for the creation of *Java SafeStrings* that are syntactically correct.

We tested each class for much longer than is normally found in the literature with the EvoSuite tool. This was motivated by the desire to understand how coverage evolves over time on Java classes before and after being June-ed. As we discovered, impact was immediate (Section 5). We also tested three other ATG tools for the more standard time of 2 minutes per class. This was done

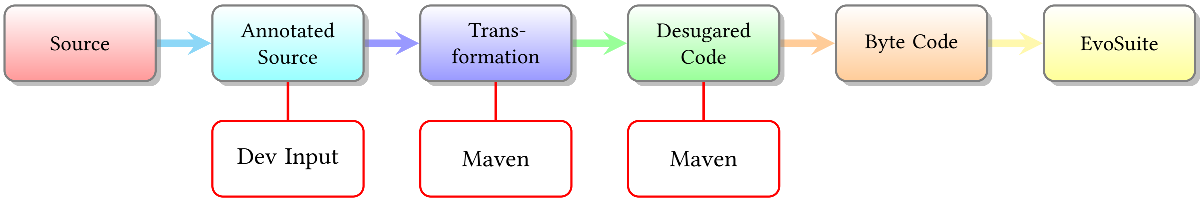

Figure 3: Typical workflow when using June to rewrite existing code for testing purposes. Only the annotation sequence requires additional developer effort. The plugin performs the program transformation and calls the compiler on the desugared code. EvoSuite has no dependency on the plugin.

to show June's tool agnosticism and that even tools which cannot generate objects can benefit from June.

4.1 Tool Selection

The natural choice of tool for testing in Java, given our preconditions, is EvoSuite [31]. This is a mature, research-driven, ATG tool founded on search-based optimisation. It uses search-based methods, implemented as different genetic techniques, including whole test suite generation, single branch strategy, and the Many-Objective Sorting Algorithm (MOSA).

We then choose other tools to demonstrate June's agnosticism, and see how June complements different testing techniques. For example, Randoop [34] is mature a testing tool, whose random techniques are powerful, albeit much simplier than EvoSuite. It requires an initial seed file for literals; it cannot generate them itself. Furthermore, it cannot generate complex objects. Selecting Randoop for our experiments, we show how simple techniques with no notion of search can see improvements from our technique.

Property based testing tools, Zest/JQF [36], and various forks of AFL [47], such as java-afl [1], mutate input files or generated input data, well-formed or otherwise. These tools are more commonly used in system testing rather than unit testing methods within a class. Nonetheless, they work at both system and unit testing levels [30]. In the hope of demonstrating June's generalisability, we wrote a small program to generate a test harness per method in order to include JQF in our experimentation. The program for writing JQF test harnesses takes each method in the class under test, wraps it in a harness method and elevates inputs to the harness. The entry point is the method under test. The final tool selected is TACO [11], another search-based testing tool based which utilises Ant Colony Optimisation in its search for inputs and sequences of method calls. It is less mature than EvoSuite, yet provides June another opportunity to show the impact it can have on a tool that uses search.

4.2 Corpus Selection

The SF110 benchmark [20] consists of 23,886 classes over 110 Java projects. SF110 builds on the SF100 corpus, which is a statistically representative sample of 100 open-source Java projects taken from SourceForge. This basis is supplemented by including the ten most popular Java projects, taken from the same source. EvoSuite is developed against the full benchmark, with results on all 23,886 classes being published with each new release [31].

To build our test corpus, we started by writing a dependency analysis, using Soot [43], to identify classes containing at least one method of interest. Here, a method of interest is one that contains at least four branches and a string input which has data flow into one or more conditional statements. This was to look for string-controlled methods. This analysis returned 2,336 methods across 1,339 classes, approximately 6% of SF110. These classes were drawn from 87 projects of the 100 projects, with an average of 12.4 branches each. This same analysis is included in June (Section 3).

We chose 500 classes, uniformly at random, for the initial design of JuneLib. We excluded these from the experiments to minimise the risk of overfitting our string types. This left 839 classes. Given our interest in studying the evolution of coverage over time, we had decided to run experiments with EvoSuite for 24hrs. We chose EvoSuite as it is the state-of-the-art in ATG for Java. Benefiting a well-designed and well-tested tool argues strongly for the power of June.

One day of computation to test one class is computationally expensive, especially given we conduct each experiment 10 times. As such, it was not possible to test all 839 classes. Instead we sampled, uniformly at random, 30 classes, which we call $SF110_{30}$. We test each class in both its original and June-ed state, resulting in 600 days of computation just for the EvoSuite experiments. Any parameters with a clear latent structure were annotated with the corresponding *Java SafeString* annotation.

4.3 Methodology and Equipment

We conducted experiments with all four tools (EvoSuite, TACO, Randoop and JQF) on both the original and June-ed classes. We annotated $SF110_{30}$ once, and reused the same code for all tools. We made no changes to any tool, and each tool ran under default settings. All experiments were run on Debian 9 with an Intel(R) Xeon(R) CPU (E5-2620 v2 @ 2.10GHz) with 10GB of allocated RAM.

Firstly, we ran each tool for 15, 60 and 120 seconds on both the original and the June-ed classes and repeated this 10 times. Common practice is that 2 minutes is enough time for unit testing ATG tools; these experiments allowed us to compare how June can assist in this small timeframe. These experiments took 32.5 hours. Then we ran EvoSuite on the corpus again but this time with a timeout of 24 hours. This choice is well above previous experimental standards (for unit testing) as found in the literature, where the 2 minute limit is more typical. For each class we completed 10 repetitions. There were a total of 60 classes (30 original and 30 June-ed), this resulted in $14,400$ hours of computation.

4.4 Threats to Validity

Our test corpus was built by uniformly sampling from a filtered version of SF110. SF110 is the standard bench mark used by EvoSuite and research using this tool [20]. We inherit all the threats to validity of the original corpus. In particular, SF110 consists of relatively old programs. There is no reason to assume, however, that the performance changes seen with June are due to older coding styles or JVM version. To validate JuneLib's construction, we also examined more recent programs from Github and found no meaningful difference.

Due to resource constraints, we were unable to test all classes in the filtered corpus. We judged 30 classes, selected uniformly at random, to be sufficient to ensure a fair sample from a filtered version of SF110. To address internal validity, we ran each experiment multiple times, as discussed in Section 4.3. June does not affect the in-class control flow of the classes, or the number of branches, so the comparison of EvoSuite's performance on original and June-ed classes is fair.

Finally, we do not claim generality of this method over all Java code. It works specifically on input strings at the method level. We sampled uniformly at random from our filtered version of SF110, and have no reason to assume that the number of string parameters is unusually low, or high *w.r.t* other Java code. The filtered corpus, for example, contains complex string parameterised functions, with at least 4 branches in the methods of interest. We do not test June on simpler methods. EvoSuite and similar tools can usually cover these shallower methods. As we show in Section 5, June allows ATG tools to achieve higher coverage, more quickly. By ignoring these simpler methods, we discount the fact that ATG tools takes at least some time to penetrate a branch, even if the string structure is simple.

5 EVALUATION

We evaluate June on 3 criteria: tool agnosticism, developer burden, and efficiency. Efficiency itself we further subdivide into effect on coverage, and effect on performance (*i.e.* speed to a coverage target). This leads to 3 research questions: RQ1) does the June testability transformation aid a variety of different ATG approaches; RQ2) how arduous is June in terms of annotation burden; and finally RQ3) does June make ATG tools more efficient, *i.e.* can they reach a coverage target more quickly? Does June increase overall coverage? The experiments conducted to answer these questions are detailed in Section 4.3.

June does not add additional branches, which is the criterion upon which we measure coverage. All comparisons are fair. There is no meaningful time overhead, as we show that time to coverage is substantially quicker on June-ed code.

5.1 RQ1: June Is Tool Agnostic

We use 4 different ATG tools to assess the universality of June-ed code. EvoSuite, Randoop, and JQF are familiar tools, well tested and well engineered. We chose a fourth tool, TACO, an ant colony optimisation based prototype, to see whether June also benefits ATG tools that are less mature in their engineering.

Figure 4 shows the coverage achieved in 2 minutes by four tools on both the original and June-ed *SF110*$_{30}$ classes. Of interest is Figure 4, showing the mean coverage achieved over a 2 minute period for each of the tools. We can see that for each allocated time, all tools perform better on June-ed classes rather than the originals. Interestingly, between 15 seconds and 2 minutes of test generation, Randoop and JQF see little increase of coverage, compared with EvoSuite which shows increasing coverage. One can attribute this to the intelligent search mechanics of EvoSuite. Even still, one can observe that June benefits all tools similarly, even enhancing the intelligent search techniques of EvoSuite, improving coverage from 46% to 51% given the 2 minute timeout. This suggests that there are areas in the classes for which it is hard to generate covering inputs. This is where June helps tools.

Looking at Figure 4, we see that the mean average coverage of *SF110*$_{30}$ improves for all tools. Randoop, EvoSuite, JQF and TACO's performance improves by 3.5, 5.6, 8.4, and 6.9 percentage points respectively, with a mean improvement of 6.1 percentage points. June achieves these gains with no modification to the tools themselves, no change in settings, configuration or environment. Instead a developer wrote 63 variable annotations and then all 4 tools benefit from the June transformation.

RQ1: June is ATG agnostic

June benefits all ATG tools tested. One can test June-ed classes with all ATG tools without making tool specific code changes.

The real power of this agnosticism is that the improvements seen by using June are complimentary to the existing tools. Any improvements to the tools are independent of the improvements provided by June. The preprocessing step can assist any ATG tool able to take advantage of it. There is no trade off between tool choice and June, and a developer can use multiple testing tools on the same June-ed classes.

5.2 RQ2: June Is Lightweight

Of key importance to the usability of June is the amount of effort required of the developer. If a tool has a negative impact on workload it is effectively useless. Effort comes in two forms for June: code annotation (Figure 2) and creating new *SafeStrings* for JuneLib (Section 3.2).

As June currently requires the user to annotate code, the number and nature of annotations is critical. Over the entirety of *SF110*$_{30}$, 63 method parameters were annotated; June's static analysis located these automatically (Section 3). Two researchers, neither expert in Java, provided the annotations. We set a five minute time budget to investigate each parameter identified by the static analysis, before either annotating or moving on. This low budget reflects time pressure on real developers in a commercial environment. Most annotations took a great deal less time than this. Meaningful parameter names, or method names which the parameter was use in, usually provided the necessary clues as to the appropriate *SafeString*. Not every string has a recognisable grammar; these we left unannotated. The final 63 annotations are just 10% of the total method parameters present in the *SF110*$_{30}$.

In total, *SF110*$_{30}$ required 11 different string annotations, all of which were already supported by JuneLib. The three most common

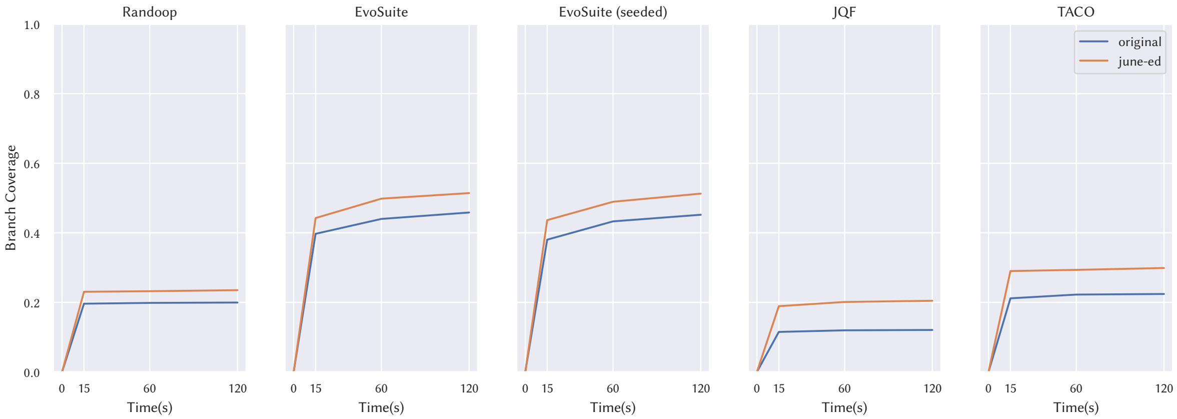

Figure 4: Average coverage over 2 minutes by (left to right) Randoop, EvoSuite, JQF and TACO. In the time budget, June-ed classes always out perform the untransformed classes.

types account for 60% of all annotations (38 of the 63 annotations were either numeric, delimited or XML).

RQ2: Cost to developer

To transform original *SF110*$_{30}$ into June-ed *SF110*$_{30}$ requires a mean average of 2.1 annotations per class. These account for just 10% of all method parameters across *SF110*$_{30}$. No new *SafeString* definitions were required.

5.3 RQ3: June's Effect on Efficiency

Efficiency comes from vastly reducing the search space over strings. ATG tools normally have no information concerning the structure of a string input. The search space is so large that, even given the sophistication of a tool like EvoSuite's mechanisms, the chances of finding the correct form of a highly structured input string are slim or require too much time. June helps with this problem for annotated strings.

We investigate June's effect on efficiency first by looking at EvoSuite's performance on *SF110*$_{30}$ over 24 hours. EvoSuite reaches 95% of max coverage in 70m 30s for the original classes (point "b" on Figure 5), impressively faster than linear. On the June-ed classes, EvoSuite completes the same task in just 5m 30s (point "a"). This massive increase in speed also comes with the fact that EvoSuite's maximum coverage is higher on June-ed classes (61.3% versus 60.3%), as shown in Figure 5. Point 'c' is the point at which June-ed classes reach greater coverage than original classes in 24 hours. One can see that EvoSuite on raw classes never achieves the same coverage as EvoSuite on June-ed classes, and June-ed classes approach the asymptote much more quickly. If coverage increased linearly one would expect to reach this milestone at 22.8 hours. However, coverage does not increase linearly; it becomes increasingly harder to 'unlock' deeper regions of code as the prerequisites become increasingly complex. June aims to solve a subset of this problem, where the prerequisite is a string input.

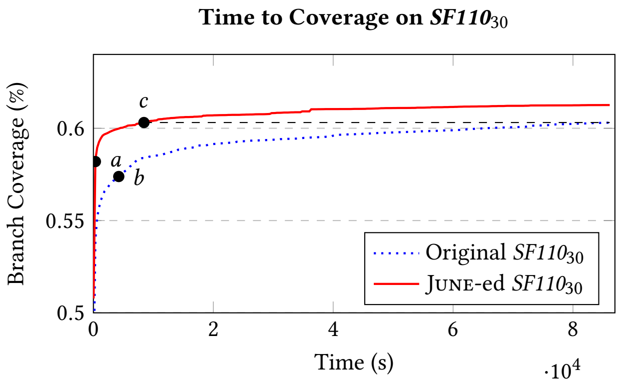

Figure 5: Change in coverage after 24 hours on EvoSuite. Original classes tested with EvoSuite do not catch up with June at any point, even after extensive testing time.

EvoSuite operates at an average rate of 10.6 percentage points of coverage per minute to reach 95% of max coverage on June-ed classes, versus 0.8 percentage points of coverage per minute on original classes. EvoSuite is clearly far more efficient on the June-ed version of classes in *SF110*$_{30}$.

RQ3a: Effect on Speed

June-ing increases EvoSuite max coverage and massively increases the rate of coverage. Max coverage on original classes is 60.3%, reaching this over 10 times more quickly on June-ed classes.

Having looked at EvoSuite's behaviour when executed for a long period of time, we now turn to look at more common execution budgets for unit test generation tools. As discussed in Section 5.1, we tested 4 different ATG tools, results for which are in Figure 4. The mean, and median, coverage achieved by all tools increased, no

Table 1: Summary statistics coverage in 2 mins, where σ is standard deviation and $\hat{A}$ is Vargha-Delaney Effect Size.

Tool / Class Type	median	mean	min	max	σ	$\hat{A}$
EvoSuite original	0.500	0.458	0.0	1.0	0.332	-
EvoSuite June-ed	0.542	0.514	0.0	1.0	0.357	0.69
EvoSuite Seed original	0.500	0.452	0.0	1.0	0.332	-
EvoSuite Seed June-ed	0.557	0.513	0.0	1.0	0.356	0.68
TACO original	0.077	0.224	0.0	1.0	0.288	-
TACO June-ed	0.125	0.293	0.0	1.0	0.338	0.60
JQF original	0.000	0.121	0.0	0.033	0.215	-
JQF June-ed	0.000	0.205	0.0	1.0	0.324	0.59
RAND original	0.094	0.200	0.0	0.700	0.220	-
RAND June-ed	0.095	0.235	0.0	0.960	0.291	0.55

minimum was increased, or maximum reduced. Summary statistics for each tool are in Table 1, including values for Vargha-Delaney Effect Size, $\hat{A}$, which has been widely recommended for use in software engineering research [5]. We also conducted a second set of experiments with EvoSuite using a seed file. This is an option available within the tool so as to leverage a pool of existing string literals. This was included to evaluate whether the benefits of June can simply be replicated by supplying instances of the structured strings. In short, they cannot. Supplied with instances of all *SafeStrings* used as annotations within *SF110*$_{30}$, EvoSuite leveraging a seed file is still improved by operating on June-ed classes rather than the originals.

Looking at the $\hat{A}$ effect size for different tools, we see values range from 0.55 for Randoop to 0.69 for EvoSuite. 0.56 is considered a small effect, 0.64 medium and 0.71 large [44]. It is therefore understandable that Randoop would receive the smallest positive effect from June. Despite providing structure for strings, Randoop itself still only provides the string "Hi!" and therefore cannot take full advantage of June. The fact that Randoop is not designed to generate complex objects and is limited to one constructor which is hard-coded into the method signature as a sequence of primitively typed parameters (Section 3) further compounds the problem.

A similar case pertains for JQF, which cannot generate objects without a generator. We did not write custom generators specifically for JQF, as JuneLib's design (Section 3.1) should make it unlikely that a developer need write any new custom *SafeString*, unless working in an environment with some custom data format. In the test harness for JQF, we used the same technique as for Randoop, changing the method signature to include primitive parameters needed for a *SafeString* constructor on the first line. TACO is a slightly smarter tool, albeit less mature, which accounts for its higher $\hat{A}$. It does attempt to search over string inputs to generate new values which cover new regions of the classes. Its $\hat{A}$ is 0.6, somewhere between small and medium effect. Finally, EvoSuite, both with and without a seed file, sees the largest effect size on June-ed classes. EvoSuite has a $\hat{A}$ value of 0.68 and 0.69 for seeded and non-seeded respectively. This is the upper end of medium effect, approaching a large effect.

Given the inability of two of the tools to generate complex objects, we investigated how effect size changes when discounting experiments that achieved zero coverage (*i.e.* failed to generate any tests). EvoSuite's $\hat{A}$ values are relatively unchanged. TACO sees an increase to 0.66, Randoop to 0.70 and JQF to 0.79. As can be seen by JQF's median coverage (Table 1), it fails to cover over half the classes. With these removed, on the remaining classes on which it does generate tests, JQF sees big gains from June. In this case, June-ing takes JQF's mean from 0.329 to 0.558 and median from 0.250 to 0.676.

For all 4 tools, coverage in 2 minutes of test generation increased by operating on June-ed classes. The average increase in mean and median was 6.1 and 3.0 percentage points respectively. Mean coverage for EvoSuite in default configuration improved by 5.6 percentage points with the help of June (1.12x the coverage on original classes). Being as complex and mature as it is, we consider this result to demonstrate the utility and power of June. Also worth noting is that both TACO and JQF saw larger improvements starting from worse initial coverage. Specifically, JQF saw mean coverage increase from 12.1% to 20.5%, a staggering 1.7x increase in performance.

RQ3b: Effect on Coverage

By operating on June-ed classes rather than the originals, EvoSuite's mean coverage of *SF110*$_{30}$ was increased by 1.12x when given 2 minutes of test generation. The average increase over all 4 tools was 1.13x.

6 RELATED WORK

Strings are a universal container, but conceptually different things are easily blurred by encoding them as strings [15]. XML is not the same as an email address, albeit both with type `String`. June permits the user to expose the conceptually diverse nature of strings to testing tools. June differs from all existing approaches in its focus on the fine-granular level of method parameters, rather than testing through a program's main entry point.

Structured Input Generation (SIG) is a problem for ATG tools [29]. The easiest approach to the SIG problem is to use randomised and search-based methods. Randoop [35] uses feedback-directed random testing over Java code. Its handling of strings, however, is rather naïve, relying on an initial seed file. If no such file is present, it uses “Hi!” as its only test input string. As Toffola *et al.* [42] argue, saying “Hi!” is not enough. Godefroid *et al.* [23] introduce a tool for *directed automated random testing*, Dart. This randomly generates strings combining symbolic and concrete executions. The search space, however, is still very large.

Many leading ATG tools, such as EvoSuite, use search-based methods for string generation [3]. Search-based SIG utilises feedback from the SUT to guide input generation. Beyene and Andrews [8] use metaheuristic techniques to generate strings for testing Java programs. They provide an automatic translation from a *context free grammar*, or CFG, to Java objects. Their work is orthogonal to June in that it considers techniques that could be of advantage to a testing tool, such as EvoSuite. Working together, one might expect to see even stronger results. Indeed, one might use their method to automatically generate new classes for JuneLib.

Grammars. The richest area of research for SIG, however, is grammar-based input generation. Maurer [33] was among the first to consider the use of an extended CFG to generate structured test input. YAGG, by Copit and Lian[14], systematically enumerates inputs from a user-provided grammar. To avoid potentially infinite string generation, the user provides bounds on the number of strings to generate.

Many popular system-testing fuzzers support some form of grammar representation to specify input formats, *e.g.*, Peach [2], Superion [45], (a grammar-aware extension to AFL [47]) and Godefroid *et al.* [22] amongst others [10]. Such grammars are typically user provided: this process is laborious, time consuming, and error-prone. Such tools require extra effort from the programmer to become effective. They allow the user to define an input DSL for a program. JUNE, on the other hand, allows the user to identify the grammar of specific variables. This is much more fine-grained approach, more suited to testing over classes rather than through a main function. In general, fuzzers require programs, *i.e.* an executable with an entry point, whereas JUNE is agnostic *w.r.t* both library and executable code. Given the empirical evidence that structured strings follow a power law (Figure 2), it is likely that the most common structured strings are already in JUNELIB.

Pythia [6] is a fuzzer that augments grammar-based fuzzing with learning-based mutation and coverage-guided feedback. Pythia still requires valid input seeds for learning. We anticipate that a JUNE annotated corpus could provide valuable information to make fuzzing with Pythia even more effective. Zest [36] uses a top-down fuzzing approach with a set of generators. These generators use grammars to create well-formed input strings. Zest requires a test harness for a program. This is a reasonable expense for Zest's principle usecase, system fuzzing. JUNE, however, is for unit testing of classes of interest. To use Zest in a manner similar to JUNE requires a test harness for each class. JUNE can have a small, re-usable library of generators for different, re-occurring string structures but they will be tied to a specific tool instead of generalising to different tools. This is a greater user effort than JUNE's annotation obligation. JUNE comes with a set of small, composable, grammars, allowing it to provide very precise information for each string parameter of a method. Complex grammars can be built through subtyping and nesting of JUNELIB objects. Zest provides no such facility. Zest, in keeping with other grammar approaches, focuses on using one large grammar (*i.e.* JSON), making it ill-suited to discovering local string types.

The manner is which Zest and JUNE generate their strings is also different. Zest's parameteric generators generate a new instance close to a previous instance by passing parameter sequences to generators. These sequences can then be mutated. On the other hand, JUNE accepts the mutable subsections of a grammar to construct a struturally valid instance by providing the immutable scaffolding required by the grammar. The goals of system and unit testing tools are different, and the required observability of how values are generated differs.

The work of Enderlin *et al.* [17] on grammar-based testing in PHP introduces PRASPEL, a formal specification language embedded into PHP. Its focus is on software validation and verification rather than unit testing. PRASPEL does not 'natively' use Java's annotation language. A standard, industry-ready, testing tool, such as EVOSUITE [12, 18–20], is all one needs to take advantage of JUNE. Ringer *et al.* [38] consider a specification language, Iorek, that provides information to an SMT solver. They use Iorek to generate strings for ATG. They show that the structure of a regex can sometimes increase code coverage. In contrast, JUNE does not require a special language to encode structural information. Generating a string that matches a regex remains difficult when the regular expression is complex. JUNE simplifies this by mixing structural elements, such as the '@' symbol in an email with more malleable substrings. We speculate that JUNE could exploit an SMT solver for string constraints to increase its effectiveness.

Learning Grammars. To increase the usability of fuzzing and SIG, recent work has focused on learning input grammars, often CFG's [7, 9, 24, 26, 27, 41]. Upon learning the grammar, the fuzzer can more easily create valid inputs to explore the program's input space. This approach is another variation on trickle-down testing. Mathis *et al.* [32] present PFUZZER, which uses a test generation technique directed at *input parsers*. This works via a feedback loop, starting from an initial seed and using information harvested from instrumentation to gather requirements. Modern approaches like Grimoire [9], or Gramatron [41], use fuzz testing to learn the input structure and then apply structure dependent mutations to create inputs. Recently, Rossouw and Fischer [39] studied the limitations introduced by grammar-based test suite construction methods, showing how they significantly bias test suites for large and real-world programs by favouring some production rules and nonterminals in a CFG over others. JUNE reduces this problem, as the *SafeString* forces randomly generated input to match local requirements, without relying on high level (and usually more complex) CFGs at some point removed from the point of use. The role of directed random string creation is thus more limited.

7 CONCLUSION

JUNE's ability to constrain the search space over structured strings substantially improves their testability in two dimensions: ATG tools are more efficient on JUNE-ed code, and the mean coverage of JUNE-ed code is higher. JUNE improves the ability of ATG tools to exercise branches that are otherwise opaque, as demonstrated on 4 commonly used ATG tools.

JUNE is a type-based testability transformation that requires minimal user effort, coming with a library of pre-defined, yet extensible, *SafeStrings*. With the low, one-off, cost of annotation, and the large efficiency boosts, JUNE substantially reduces the cost of test suite creation and maintenance for string-manipulating programs.

ACKNOWLEDGMENTS

Kelly and Menéndez were both partially supported by the UKRI Trustworthy Autonomous Systems Hub (reference EP/V00784X/1) and Trustworthy Autonomous Systems Node in Verifiability (reference EP/V026801/2). Barr was partially supported by DAASE (EP/J017515/1). Clark was partially supported by the InfoTestSS project (EP/P005888/1).

REFERENCES

[1] [n. d.]. java-afl. https://github.com/Barro/java-afl. Accessed: 2022-11-11.
[2] Accessed: 2022-11-11. Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/.

[3] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Softw. Test., Verif. Reliab.* 16 (09 2006), 175–203. https://doi.org/10.1002/stvr.354

[4] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd ed.). Cambridge University Press, New York, NY, USA.

[5] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.

[6] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. (05 2020).

[7] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

[8] Michael Beyene and James H. Andrews. 2012. Generating String Test Data for Code Coverage. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 270–279. https://doi.org/10.1109/ICST.2012.107

[9] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[11] Dan Bruce, Héctor D Menéndez, Earl T Barr, and David Clark. 2020. Ant colony optimization for object-oriented unit test generation. In *International Conference on Swarm Intelligence*. Springer, 29–41.

[12] José Campos, Annibale Panichella, and Gordon Fraser. 2019. EvoSuite at the SBST 2019 Tool Competition. In *Proceedings of the 12th International Workshop on Search-Based Software Testing (SBST '19)*.

[13] Carl Chapman and Kathryn T. Stolee. 2016. Exploring Regular Expression Usage and Context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/2931037.2931073

[14] David Coppit and Jiexin Lian. 2005. Yagg: An easy-to-use generator for structured test inputs. *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, 356–359. https://doi.org/10.1145/1101908.1101969

[15] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T Barr. 2018. RefiNym: using names to refine types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 107–117.

[16] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

[17] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Fabrice Bouquet. 2012. Grammar-Based Testing using Realistic Domains in PHP. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012* (04 2012). https://doi.org/10.1109/ICST.2012.136

[18] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

[19] Gordon Fraser and Andrea Arcuri. 2013. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[20] Gordon Fraser and Andrea Arcuri. 2014. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 8.

[21] Gordon Fraser, José Miguel Rojas, and Andrea Arcuri. 2018. Evosuite at the SBST 2018 Tool Competition. In *Proceedings of the 11th International Workshop on Search-Based Software Testing* (Gothenburg, Sweden) *(SBST '18)*. Association for Computing Machinery, New York, NY, USA, 34–37. https://doi.org/10.1145/3194718.3194729

[22] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2007. *Grammar-based Whitebox Fuzzing*. Technical Report MSR-TR-2007-154. 10 pages. https://www.microsoft.com/en-us/research/publication/grammar-based-whitebox-fuzzing/

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *In Programming Language Design and Implementation (PLDI*.

[24] Rahul Gopinath, Björn Mathis, Mathias Höschele, Alexander Kampmann, and Andreas Zeller. 2018. Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing. *arXiv preprint arXiv:1810.08289* (2018).

[25] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering* 30, 1 (2004), 3–16.

[26] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

[27] Matthias Höschele and Andreas Zeller. 2017. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 31–34.

[28] David Kelly, Mark Marron, David Clark, and Earl T. Barr. 2019. SafeStrings: Representing Strings as Structured Data. *CoRR* abs/1904.11254 (2019). http://arxiv.org/abs/1904.11254

[29] B. Korel. 1990. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.* 16, 8 (Aug. 1990), 870–879. https://doi.org/10.1109/32.57624

[30] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[31] EvoSuite Maintainers. 2022. EvoSuite: Automatic Test Suite Generation for Java. https://www.evosuite.org/

[32] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 548–560.

[33] Peter M. Maurer and Peter M. Maurer. 1990. Generating test data with enhanced context-free grammars. *IEEE Software* 7 (1990), 50–55.

[34] Carlos Pacheco, Shuvendu Lahiri, Michael D. Ernst, and Tom Ball. 2006. *Feedback-directed Random Test Generation*. Technical Report MSR-TR-2006-125. 14 pages. https://www.microsoft.com/en-us/research/publication/feedback-directed-random-test-generation/

[35] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *In ICSE*. IEEE Computer Society.

[36] 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 (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[37] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[38] Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. 2017. A Solver-aided Language for Test Input Generation. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 91:1–91:24. https://doi.org/10.1145/3133915

[39] Christoff Rossouw and Bernd Fischer. 2021. Vision: bias in systematic grammar-based test suite construction algorithms. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. 143–149.

[40] M. Sipser. 2012. *Introduction to the Theory of Computation*. Cengage Learning. https://books.google.co.uk/books?id=X0puCgAAQBAJ

[41] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 244–256.

[42] L. D. Toffola, C. Staicu, and M. Pradel. 2017. Saying 'Hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 44–49. https://doi.org/10.1109/ASE.2017.8115617

[43] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[44] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[46] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. 2021. Characterizing the Usage, Evolution and Impact of Java Annotations in Practice. *IEEE Transactions on Software Engineering* 47, 5 (2021), 969–986. https://doi.org/10.1109/TSE.2019.2910516

[47] Michal Zalewski. 2007. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

Received 2023-02-16; accepted 2023-05-03