Generating Conforming Programs With Xsmith

William Gallard Hatch
william@hatch.uno
University of Utah
USA

Pierce Darragh
pierce.darragh@utah.edu
University of Utah
USA

Sorawee Porncharoenwase
sorawee@cs.washington.edu
University of Washington
USA

Guy Watson
guy.watson@utah.edu
University of Utah
USA

Eric Eide
eeide@cs.utah.edu
University of Utah
USA

Abstract

Fuzz testing is an effective tool for finding bugs in software, including programming language compilers and interpreters. Advanced fuzz testers can find deep semantic bugs in language implementations through differential testing. However, input programs used for differential testing must not only be syntactically and semantically valid, but also be free from nondeterminism and undefined behaviors. Developing a fuzzer that produces such programs can require tens of thousands of lines of code and hundreds of person-hours. Despite this significant investment, fuzzers designed for differential testing of different languages include many of the same features and analyses in their implementations. To make the implementation of language fuzz testers for differential testing easier, we introduce Xsmith.

Xsmith is a Racket library and domain-specific language that provides mechanisms for implementing a fuzz tester in only a few hundred lines of code. By sharing infrastructure, allowing declarative language specification, and by allowing procedural extensions, Xsmith allows developers to write correct fuzzers for differential testing with little effort. We have developed fuzzers for several languages, and found bugs in implementations of Racket, Dafny, Standard ML, and WebAssembly.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Compilers**.

Keywords: automated testing, compiler testing, fuzzing, random program generation, random testing

ACM Reference Format:
William Gallard Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs With Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3624007.3624056

This work is licensed under a Creative Commons Attribution 4.0 International License.
GPCE '23, October 22–23, 2023, Cascais, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0406-2/23/10.
https://doi.org/10.1145/3624007.3624056

1 Introduction

The effectiveness of random testing, or "fuzzing," is determined both by the chosen input generation strategy and the method used to detect failing tests, or *test oracle*. The generation or mutation of test cases using random bytes can in theory generate any test and therefore cover any code path, but typically exercises only "shallow" code in parsing and input validation stages of a program. Meanwhile, grammar- and type-aware generators can exercise "deep" code paths that pass validation steps. The test oracle of detecting crashes can be used with any test case generator, but can only find obvious errors such as memory or assertion violations, and not subtle semantic bugs. Property-based testing can find semantic bugs, but requires users to write invariant properties of test results or side effects, which is expensive.

The test oracle of interest for this paper is *differential testing* [17], where the same input is given to multiple implementations of a system—in our case, a programming language. If the multiple implementations are correct, then giving them all the same input program (and executing the returned output if the implementation is a compiler) should produce the same result. When there is a difference in program output, a bug has been found. While differential testing can find subtle semantic bugs without writing extra properties, there is a catch. If the input program relies on any behavior that is not guaranteed to be the same between multiple executions and multiple implementations of the language, differences in output do *not* necessarily indicate a bug. Therefore, differential testing requires programs that conform strictly to the language specification, as well as avoiding undefined, implementation-defined, or nondeterministic behavior. We call such inputs *conforming* inputs.[1]

One notable generator of conforming programs is Csmith [24]. Csmith successfully identified hundreds of bugs

[1]We considered other words to convey that a program is suitable for differential testing, such as the word *differentiable*. However, the word *differentiable* has other meanings that would make this confusing.

$$\begin{array}{lcl} \langle int \rangle & ::= & z \mid 0 < z < 100 \\ \langle exp \rangle & ::= & \langle exp \rangle \; + \; \langle exp \rangle \\ & \mid & \langle exp \rangle \; / \; \langle exp \rangle \\ & \mid & \langle int \rangle \end{array}$$

Figure 1: The Grammar of the `calc` Language

in mainstream C compilers, including LLVM and GCC. However, the development of Csmith took hundreds of person-hours and resulted in nearly 40,000 lines of code. Although practically any language could benefit from such a program generator, it is impractical for most language developers to write a variant of Csmith targeting their own language.

To ease the development of fuzzers that generate conforming programs, we created *Xsmith*. Xsmith is a domain-specific language (DSL), implemented as a Racket library, that allows for the rapid and concise implementation of conforming program generators for arbitrary programming languages.[2]

The primary contributions of this paper are:

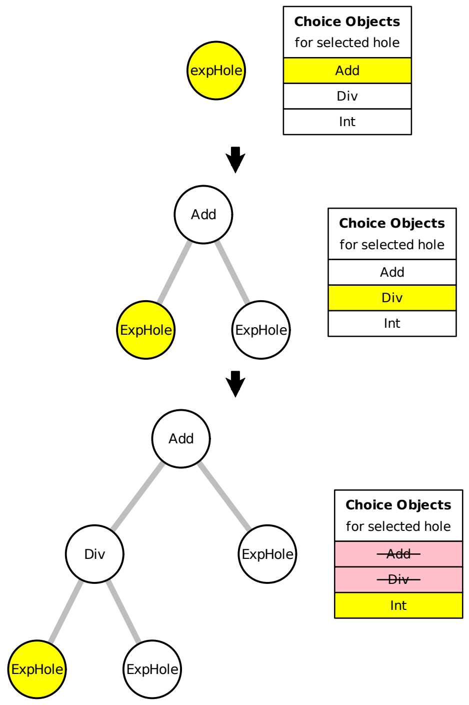
- A domain-specific language for creating conforming program generators.
- A generic framework for declaring type and effect systems for program generation used in the DSL implementation.
- Several example fuzzers implemented using the DSL, some of which have been used to find bugs in language implementations used in production.
- An analysis of the effectiveness and cost of the example fuzzers.

2 Design

In this section we describe the overall design of Xsmith at a high level. Throughout the section, we develop a simple fuzzer for a small toy calculator language (named `calc`) as an example. The grammar of this language is shown in Figure 1.

Xsmith fuzzers generate program trees by starting with a "hole" node for the top-level production of the grammar. Xsmith iteratively fills hole nodes in the tree with nodes corresponding to appropriate grammar productions, which may themselves have holes as children, as shown in Figure 2. A fuzzer author provides Xsmith with a grammar declaration that determines the grammar of program generation.

The grammar used by an Xsmith fuzzer generally matches the logical structure of the language or a subset of the language, but is usually not the same as the grammar used for parsing the language or the grammar of a compiler's AST representation. This is because parser grammars typically encode complicated rules for turning linear text into an AST, and because full language grammars can be large and difficult to model for producing conforming programs. Xsmith's



The tree starts as a single `ExpHole` node. Choices that are alternates for `Exp` are listed and one is chosen at random. The process repeats with a new hole until no holes are left. At some points choices are filtered. For example, non-atomic choices are filtered when the tree gets too deep.

Figure 2: The Process of Hole Filling

simplified AST-focused grammar system allows users to ignore syntax complexity to focus on modeling semantics for finding "deep" bugs. Users can start with a small subset of the grammar, and iteratively grow their subset to include more features.

The declared grammar is compiled to generate an object-oriented attribute grammar specification for the RACR attribute grammar library [3]. RACR facilitates analysis of generated program fragments, allowing data flow both up and down the tree for any analysis. The generated attribute grammar matches the input grammar but adds a hole production as an alternative for each user-provided production.

While Xsmith program generation is grammar-driven at its core, generation of conforming programs requires considerations besides the grammar, such as for types and nondeterminism. To enable filtering and probability weighting based on these other considerations, the grammar is also compiled into a set of *choice classes*, one for each given production. When the generation algorithm selects the next hole to fill, a choice object is instantiated for each production that could be used to replace the hole, and the resulting list of choice objects is used to make the decision.

[2]Xsmith is open source, with code available at https://gitlab.flux.utah.edu/xsmith/xsmith/

The difference between the attribute grammar and choice classes is often confusing to people learning about Xsmith for the first time. During generation, the AST being generated is represented by attribute grammar nodes, including hole nodes. Choice objects, or instances of choice classes, are not part of the AST, but are constructed when considering a particular hole node to fill. Each generated choice object corresponds to a different AST node choice that could be used to replace the currently chosen hole node, according to the grammar. Choice class methods are used to determine whether the given production choice is appropriate for the hole, given constraints besides the grammar, such as for types or unspecified behavior. To make this determination, choice class methods may query attributes of the AST, as well as other choice class methods on the same choice object. When a particular choice is made, the *fresh* method of the chosen choice object is used to construct a new attribute grammar node to replace the hole node. Then generation either moves on to another hole, which constructs new choice objects, or terminates.

Below is an example Xsmith specification that implements the grammar of the `calc` language. Each arm of the `add-to-grammar` form contains the name of a production, the supertype of that production, and a list of fields for that production. The `Add` and `Div` productions each have two children, and these children must themselves be `Exp` (expression) productions. These `Exp` children will be instantiated as `Exp` holes. The `Int` production specifies an integer literal, whose `val` child may contain arbitrary Racket data. In this example, `val` is initialized to a random integer between 1 and 100. (This random selection occurs each time an `Int` hole is filled.)

```
;; Fuzzers may use multiple add-to-grammar forms to
;; declare a grammar in a modular fashion.
(add-to-grammar calc
 [Exp #f ()
      #:prop may-be-generated #f]
 [Add Exp ([lhs : Exp]
           [rhs : Exp])]
 [Div Exp ([lhs : Exp]
           [rhs : Exp])]
 [Int Exp ([val = (random 1 100)])])
```

In addition to writing a grammar, a fuzzer author can declare various *attributes* and *choice methods* of each grammar production.

During generation, an Xsmith-based fuzzer uses choice methods to determine whether a particular choice of grammar production is suitable for replacing a hole. For example, the `_xsmith_satisfies-type?` choice method is used to filter out choices with invalid types, the `_xsmith_wont-over-deepen` method is used to make choices that will keep the generated program size bounded, and the `_xsmith_fresh` choice method determines how a chosen node is initialized.[3] Choice methods may call other choice methods, as well as attributes of their corresponding hole node.

Attributes are methods of AST nodes, and are used to perform analysis. For example, the `xsmith_type` attribute computes the type of a node, used by choice methods such as `_xsmith_satisfies-type?`, and the `_xsmith_visible-bindings` attribute computes a list of bindings available for reference at a given point in the tree. Attributes may query other attributes on the same AST node or on other nodes during computation, allowing data to flow up and down the tree as needed to make choices.

Besides attributes and choice methods, users may also declare *properties* of each grammar production. Properties are an abstraction on top of attributes and choice methods which simplify the specification of many aspects of a language. Each property is essentially a small custom DSL for describing an aspect of a language. Properties are compiled into attributes and choice methods. While choice methods and attributes may be written directly in a procedural style, properties allow semantic details of a grammar production, such as their binding structure and types, in a declarative style. Properties range from being convenient syntax sugar for defining a single attribute to being complicated DSLs that analyse multiple properties together to generate a family of attributes and choice methods.

Typical Xsmith fuzzers describe most language details with properties rather than defining many attributes or choice methods directly. For example, the `type-info` property described in section 4.2 is used to generate both the `xsmith_type` attribute and the `_xsmith_satisfies-type?` choice method. A collection of properties is included with Xsmith for describing key features of a language. Users may additionally define custom properties with their own compilation transformers to abstract patterns between fuzzers, although Xsmith is designed to support the majority of properties most languages' fuzzers would need out of the box.

Properties may be declared inline with a grammar definition, as with the `may-be-generated` property used in the `calc` grammar shown above, or separately with the `add-property` form. In the following code, `choice-weight` of the `Div` and `Int` nodes is adjusted to change their generation probabilities.

```
(add-property calc choice-weight
 [Div 10]
 [Int 5])
```

When a tree with no holes is finally completed, it must be converted to text for programming language implementations to consume. The `xsmith_render-node` attribute, defined by the `render-node-info` property, is used to convert the tree into text. Below is an example renderer for our `calc` language.

[3]Because attribute and method names are bare symbols without namespacing, we use the `xsmith_` prefix by convention for names defined by Xsmith itself, and we use a leading underscore by convention for private attributes and methods intended for use only in Xsmith's implementation.

```
(define (render-infix operator)
  (lambda (n)
    (format "(~a ~a ~a)"
            (att-value 'xsmith_render-node
                       (ast-child 'lhs n))
            operator
            (att-value 'xsmith_render-node
                       (ast-child 'rhs n)))))
(add-property calc-grammar render-node-info
 [Add (render-infix "+")]
 [Div (render-infix "/")]
 [Int (lambda (n)
        (number->string (ast-child 'val n)))])
```

2.1 Validating The Grammar

Because users define their own subset of a grammar rather than using published parser grammars, and because they must encode many rules about the language semantics, a user may write a generator that is incorrect. While writing a generator, users can cross-validate it by running generated test cases against implementations of the language, or systems under test (SUT). There are four major ways in which an Xsmith-based fuzzer may be incorrect:

- Test cases produced by Xsmith are syntactically or statically semantically invalid (e.g., with static type errors). In this case, the SUT will reject them, often with helpful error messages.
- Test cases are dynamically invalid (e.g., triggering runtime type errors). A user would also observe errors when interacting with the SUT.
- Generated test cases execute invalid or nondeterministic behaviors. Such cases can often be detected through differential testing of multiple implementations or through compile-time sanitizers.
- The grammar and associated generation rules provided to Xsmith describe only a fraction of the language. In this case, Xsmith would generate conforming test cases, but without leading to good coverage of implementations and/or bug-finding power. A user can discover this by inspecting generated test cases and the obtained coverage of the SUT following a fuzzing campaign.

3 Example

To give a sense of what a small but still featureful Xsmith fuzzer looks like, we present a small JavaScript fuzzer. Figure 3 is an abbreviated example of a simple JavaScript fuzzer, with elided code sections marked by "...". A full version of this example is included in the Xsmith source repository. While the full version is longer, it is still only 412 lines as measured by the wc utility.

This example demonstrates a fuzzer that takes advantage of the canned-components library to generate conforming JavaScript programs that may utilize arrays, first-class functions, objects (encoded in Xsmith as structural record types), if statements, and loops. The largest amount of code elided from the full version is in program rendering. The rendering step tends to be verbose and varies by language, but is not complicated or difficult to code.

This example uses a `safe_divide` function, defined in a header, to avoid issues that arise from dividing by zero. Similarly, the full version defines more safe wrappers for array reference and assignment. While these operations are not undefined or even necessarily troublesome behavior in JavaScript, we use them to avoid having values collapsing to JavaScript's `undefined` value, which would otherwise be overwhelmingly common. This demonstrates a common pattern used when creating fuzzers with Xsmith to avoid undefined behavior or the raising of common exceptions.

This example also does not directly use any `add-to-grammar` forms, because the entire abstract grammar used is provided by the canned components library, discussed in section 4.3. A larger fuzzer will generally include canned components as well as `add-to-grammar` forms that add various built-in functions specific to the language.

4 Cost Reduction Features

Xsmith has many features that work together to make the creation of conforming program generators inexpensive in terms of development time and effort. These features include forms for declaring grammar, types, and name scoping and resolution, as well as a "canned components" library to encapsulate language similarities, features for undefined behavior handling, and so on. In this section we give an overview of these features, discussing their usage and design.

4.1 Grammar and Syntax

The first step to generating programs that are conforming is to follow a grammar. Xsmith generates program trees according to a grammar provided by the user.

Usage. A user can define a grammar for their generator by using the `add-to-grammar` form. Each grammar production is declared as a subtype of another grammar production (possibly the abstract base grammar production, referenced by `#f`). Grammar productions may have any number of children, which can be specified as either being grammar productions (of a given type), or storage locations for arbitrary Racket data (used, for example, to hold values for number literals). Children may be annotated with a Kleene star to indicate repetition (zero or more repetitions). Below is an example of a partial grammar definition.

```
(add-to-grammar js-component
 [ArrayLiteral Expression ([elem : Expression *])]
 [IntLiteral Expression ([value])])
```

Design and Implementation. Xsmith's grammar and AST data structures rely on the RACR [3] attribute grammar library. RACR allows Xsmith grammar nodes to have attributes that compute data that can depend dynamically

```
(require xsmith xsmith/canned-components racr pprint ...)

;; An Xsmith specification starts with a "spec-component"
(define-basic-spec-component js-component)

;; Use canned-components to get common grammar definitions.
(add-basic-expressions js-component
                        #:LambdaWithBlock #t
                        #:MutableArray #t
                        ...)
(add-basic-statements js-component
                       #:ProgramWithBlock #t
                       #:IfElseStatement #t
                       ...)

;; Use canned-component loop generator.
;; It has many options, some elided.
(add-loop-over-container js-component
                          #:name ForLoopOverArray
                          #:loop-ast-type Statement
                          #:body-ast-type Block
                          #:collection-type-constructor
                          (λ (elem-type)
                            (mutable (array-type elem-type)))
                          ...)

;; This header defines safe wrapper operations, and is included
;; when rendering the program.
(define header-definitions-block
  "function safe_divide(a,b){return b == 0 ? a : a / b} ...")

(add-property js-component render-node-info
              [VariableReference
               (λ (n) (text (ast-child 'name n)))]
              [SafeDivide
               (λ (n) (h-append
                       (text "safe_divide(")
                       (render-child 'l n)
                       (text ", ")
                       (render-child 'r n)
                       (text ")")))]
              [IfElseStatement
               (λ (n)
                 (h-append
                  (text "if (")
                  (render-child 'test n)
                  (text ")")
                  (render-child 'then n)
                  (text " else ")
                  (render-child 'else n)))]
              ...)

;; This macro defines, among other things, a function
;; to run the command line parser and start
;; generation with the given parameters.
(define-xsmith-interface-functions [js-component]
  #:program-node ProgramWithBlock
  #:format-render (λ (doc) (pretty-format doc 120))
  ...)
```

Figure 3: Sample JavaScript Generator Written with Xsmith

on attributes or data from parent or child nodes. As an AST grows, RACR automatically keeps track of which attributes need to be recomputed.

Xsmith relies on grammars to allow users to define and re-use language components. To transform the AST into an input that is syntactically valid for a compiler or interpreter, Xsmith includes a multi-step render phase. The goal of the render phase is to produce a program as text. However, rather than defining a transformation from each grammar node to a string, it can be easier to have pleasantly formatted output by using an intermediate format. For example, the AST can be rendered into the data structures of Racket's `pprint` pretty printing library or to s-expressions, which both have pretty-printing functions available.

4.2 Types

Generators of conforming programs need to produce well-typed code to pass the type-checking stage of programming language implementations. The requirement for type-correct code is perhaps less strict for dynamically typed languages than for statically typed languages. However, if code for dynamically typed languages is generated without regard to types, most expressions will raise run-time type errors. Xsmith includes a type system framework that allows fuzzers to generate well-typed code for a variety of languages.

Usage. The type system used by a fuzzer is defined by the `type-info` property. This property takes two specifications per grammar node. The first specification defines the types a grammar node can inhabit. The second specification is a function that receives a tree node of the specified production and its type and returns a dictionary of types for the node's children. In the code below, the `LiteralString` and `StringAppend` productions are declared to always have type `string`, while the `VariableReference` is declared to use a type variable that can be unified with any type. The `LiteralString` and `VariableReference` productions have no children, but the `StringAppend` production constrains its children to inherit its type.

```
(define no-child-types (lambda (n t) (hash)))
;; The `hash` function constructs a hash table
(add-property
 js-component
 type-info
 [LiteralString [string no-child-types]]
 [StringAppend [string (lambda (n t) (hash 'l t 'r t))]]
 [VariableReference [(fresh-type-variable)
                     no-child-types]])
```

Design and Implementation. Xsmith allows its user to specify type systems that contain base types, function types, product types, generic types (such as lists and arrays), nominal records, and structural records. Xsmith supports less expressive type systems than some other tools do, such as PLT Redex [6]. Some of these tools, like Redex, support arbitrary type judgments that are compiled to first-order logic and subsequently used for type checking and generating random terms [7]. However, Xsmith has more constraints on generated programs than merely being well-typed. Other analyses, such as those for the effect system described in section 4.4, require structural reasoning on types. Therefore, we have built a more limited type definition framework that allows this cross-analysis. Despite these limitations, Xsmith's type checking framework is sufficient to support fuzzing many features of popular programming languages.

Type systems specified in Xsmith may also support subtyping. During type checking, Xsmith performs *subtype unification* between the types that a tree node declares that it supports, the types provided by its parent node, and any types declared by relevant definition nodes for references. Subtype unification, like traditional variable unification during type inference, mutates type variables to indicate relationships between type variables and between type variables and concrete types. However, unlike traditional unification, subtype unification reflects the asymmetric relationship of subtyping. Type variables in subtype relationships form a lattice of related types, where `(subtype-unify! a b)` relates `a` as a subtype of `b`, placing `a` below `b` in the lattice. Symmetric unification in this model is implemented merely as two subtype unifications:

```
(define (unify! a b)
  (subtype-unify! a b)
  (subtype-unify! b a))
```

When a type variable `a` is already related as a subtype to type variable `b` and `(subtype-unify! b a)` is executed, the relationship lattice is squashed such that `a`, `b`, and all variables between them in the lattice are unified into a single type variable.

While it is well known that unification-based type inference is incompatible with subtyping for type checking of existing programs, Xsmith can use unification because it type checks program fragments while generating a fresh program. So, assuming the type system specification is correct, for any needed type, it is always possible to produce a term of that type.

4.3 Language Similarities

Many programming languages have similar language features. To help Xsmith users avoid implementing these features afresh for each language they write an Xsmith fuzzer for, Xsmith provides a library of "canned components" that automatically define the necessary grammar nodes and properties.

Usage. The main forms provided by the library are the `add-basic-statements` and `add-basic-expressions` macros. Each of these has a variety of optional keyword arguments and extends a grammar with forms specified by those arguments. These productions include literals, accessors and mutators for mutable arrays and dictionaries, and function application and definition. The code below demonstrates how many standard productions can be added to a grammar, with appropriate type rules and other properties, with a `canned-components` macro.

```
(add-basic-statements js-component
                      #:Block #t
                      #:ReturnStatement #t
                      #:IfElseStatement #t
                      #:AssignmentStatement #t
                      ...
                      )
```

The `canned-components` library also provides the `add-loop-over-container` macro, which has various keyword arguments allowing a user to specify whether the loop form is a statement or an expression, which types of containers it can loop over, and the type of the loop's result. These productions are added with all relevant properties for the type and effect systems, name analysis, etc. The only non-optional property that the user must add is the `render-node-info` property. The code below demonstrates how a loop form can be added to a grammar.

```
(add-loop-over-container js-component
 #:name ForLoopOverArray
 #:loop-ast-type Statement
 #:body-ast-type Block
 #:collection-type-constructor
 (λ (elem-type) (mutable (array-type elem-type)))
 ...)
```

Design and Implementation. The canned components are implemented as a library of macros that generate the most common patterns of grammar and property definitions. The `canned-components` library reduces the amount of code required to write a new fuzzer, and it reduces the duplication of tedious and error-prone type rules and other properties that are easy to get slightly wrong.

4.4 Unspecified and Implementation-Defined Behavior

For practical reasons, some programming languages leave the semantics of certain constructs either up to individual implementations or completely undefined. A generator of conforming programs must avoid every type of unspecified or non-deterministic behavior in the programs that it generates. One common unspecified behavior concerns the order of evaluation of subexpressions, such as multiple arguments in a function call. While the evaluation order is unimportant in the evaluation of purely functional code, effectful code that assigns variables or mutates values requires a consistent ordering to be conforming.

Usage. To avoid generating code with an unspecified effect order, a user simply annotates which nodes include different effects, such as reading and writing to variables or mutable data structures. This is demonstrated in the following code.

```
(add-property js-component reference-info
    [Reference (read)]
    [Assignment (write)])
```

Design and Implementation. Xsmith includes an effect analysis that enumerates the possible effects of code evaluation and conservatively avoids ordering conflicts. Tracked effects include variable reference and assignment, projection and mutation of values like arrays, and higher-order function application. Whenever a potential conflict arises, such as referencing a variable in one function argument while assigning to the same variable in another argument, Xsmith filters out the choices that would lead to the generation of offending programs. A user may annotate grammar nodes that impose a specified ordering on their children, such as block and sequence constructs, with the `strict-child-order?` property.

Besides effect ordering, programming languages have an inconsistent variety of features that cause undefined, or at least unhelpful, behavior. For example, out-of-bounds array access is an undefined behavior in C, while in many other languages it is defined to raise an exception. Although a raised exception is well defined and potentially an interesting part of the language API to fuzz test, in typical fuzz testing an array access exception is likely not a useful behavior. For example, because the set of possible values of type `int` in a given programming language is likely much larger than the set of usable array sizes, array access with approximately uniformly generated `int` values will raise exceptions much more often than it will yield values. For both defined and undefined semantics of array access, it is usually best to generate code that wraps such accesses to convert the index to a number within bounds or provide a fallback result value.

Since these behaviors are
language-specific, each fuzzer needs some amount of unique attention to them. The common pattern used in our example fuzzers is to include program header text that defines safe wrapper functions for potentially problematic functionality, possibly including extra fallback arguments (e.g., for accessing an empty list). The grammar can then target the safe wrappers instead of the raw unsafe operations. Some of these behaviors are common and have been captured in the canned-components library.

4.5 Name Scoping and Resolution

Generators of conforming programs need to produce programs where variables referenced are defined in scope. Xsmith includes a generic analysis to ensure that variables are well scoped. If a reference is generated in a position where no appropriately typed variable is in scope, Xsmith will automatically add an appropriate definition node into a scope that is visible to the new reference.

Usage. Users can annotate which grammar nodes bind and reference variables using the `binder-info` and `reference-info` properties, such as with the following code.

```
(add-property js-component binder-info
  [Definition ()]
  [FormalParam (#:binder-style parameter)])
```

However, common patterns for binders and references have also been captured in the canned-components library, so most users do not need to interact with these properties directly.

Design and Implementation. Our resolution system is based on scope graphs [18], which is a generic system for representing variable scoping in programming languages. Based on user-provided annotations, Xsmith will generate scope graph models for generated programs, and use them to find which variables that are in scope at any position.

4.6 Language-Specific Analyses

While Xsmith includes several generic analyses, an advanced fuzzer may benefit from a language-specific analysis. Because they are language-specific, such analyses can not reasonably be included in the Xsmith framework. However, Xsmith provides features that aid a user in writing custom analyses.

Users can define custom attributes and choice methods, as well as custom Xsmith properties. Custom properties are essentially mini-DSLs that can compile declarative data into attributes and choice methods. Defining custom properties requires familiarity with Racket macro writing techniques, and we will leave discussion of custom properties to the Xsmith documentation. Finally, users can leverage Xsmith's generic analyses as dependencies of their analyses, such as by querying a node's type during a custom analysis.

Custom properties, attributes, and choice methods provide a way for Xsmith users to extend Xsmith with arbitrary Racket code. This allows Xsmith fuzzers to include features never imagined by Xsmith's authors.

4.7 Making Decisions

Xsmith includes features for both filtering potential decisions and for adjusting the probability of different choices when filling holes in the generated AST.

Usage. A user can add custom choice methods as filters by using the `choice-filters-to-apply` property. The following code applies filter `choice-method` defined above to restrict `VariableReference` generation.

```
(add-property choice-filters-to-apply js-component
  [VariableReference (allow-ref)])
```

A user can adjust the frequency of different grammar node choices with the `choice-weight` property, shown below.

```
(add-property choice-weight js-component
              [IfStatement 50]
              [AssignmentStatement
               (λ (hole) (if (eq? (ast-node-type
                                   (ast-parent hole))
                                  'IfStatement)
30))])
```

The choice weight may be given a positive integer or a function that returns a positive integer based on an analysis of the program.

Design and Implementation. When filling a hole, Xsmith instantiates one choice object with the appropriate class for each subtype of the required node type. For example, in a hole of type `Expression`, Xsmith will instantiate a choice object for each of `IntegerLiteral`, `VariableReference`, `Addition`, and so on. Each of these choices is filtered based on the specifications given to the `choice-filters-to-apply` property, including default filters such as type satisfaction. After a list of valid choices has been filtered, each remaining choice has its `choice-weight` computed. A choice is then randomly made, with each choice having probability *choiceWeight / weightSum*.

4.8 Additional Features

Xsmith includes various other features that we lack room to discuss, such as an integrated automatic test-case reducer, an extensible command-line interface, support for modular fuzzer declaration, iterative refinement, parametric generation in the manner of Zest [20], and more.

5 Evaluation

We evaluate Xsmith by considering a set of fuzzers built with Xsmith as well as bugs found with those fuzzers. We assess the difficulty of creating fuzzers with Xsmith and the number and quality of bugs found. In particular, we examine a particular case study of fuzzing Dafny.

5.1 Fuzzers

Generators of conforming programs typically require a lot of effort to create. Csmith, a predecessor to Xsmith and its major inspiration, required hundreds of person-hours and tens of thousands of lines of code. Xsmith fuzzers require substantially less effort and code. Figure 4 compares sizes of a selection of conforming program generators in terms of code size.

While the implementations of Xsmith-based fuzzers are significantly smaller than similar conforming program generators like Csmith [24], Verismith [10], and SQLSmith [21], they still produce programs that are syntactically and semantically valid as well as free from undefined or nondeterministic behavior. Additionally, Xsmith fuzzers can be featureful, generating correct code for conditionals, rich types, variable references, and so on.

Xsmith is not the only generic framework for creating programming language fuzzers. Other generic frameworks include Polyglot [4] and StarSmith [13]. While Xsmith has the greatest focus on differential testing compared to other generic frameworks, it compares well in terms of implementation effort per fuzzer, as shown in Figure 5.

Generator	**LOC**	**Language**
Csmith	38,988	C++
Verismith	10,139	Haskell
SQLsmith	3,909	C++
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket

All line counting was done with Unix `wc`. Fuzzers marked with * have not been exercised in substantial fuzzing campaigns.

Figure 4: Comparison of Conforming Program Generators

Generator	**LOC**	**Language**
Xsmith Framework	13,325	Racket
Xsmith Racket Fuzzer	1,265	Racket
Xsmith Dafny Fuzzer	1,666	Racket
Xsmith Standard ML Fuzzer	1,151	Racket
Xsmith WebAssembly Fuzzer	1,433	Racket
Xsmith Python Fuzzer *	1,800	Racket
Xsmith Lua Fuzzer *	450	Racket
Xsmith Javascript Fuzzer *	412	Racket
StarSmith Framework	19,524	Java
StarSmith C Fuzzer	1,702	LaLa
StarSmith Lua Fuzzer	1,578	LaLa
StarSmith SQL Fuzzer	~ 3,500	LaLa
StarSmith SMT Fuzzer	~ 700-900	LaLa
Polyglot Framework		Mostly C++
Polyglot C Fuzzer	1,508	Mix
Polyglot JavaScript Fuzzer	1,618	Mix
Polyglot PHP Fuzzer	2,013	Mix
Polyglot Solidity Fuzzer	2,090	Mix

Counts prefixed with ~ are approximate. Fuzzers marked with * have not been used in substantial fuzzing campaigns.

Figure 5: Comparison of Generic Fuzzing Frameworks

Some StarSmith line counts are approximate, because for SQL and SMT their repository contains multiple versions of each fuzzer with various modifications. The size of the Polyglot framework is difficult to ascertain, as the bulk of its implementation is a modification to AFL, and the Polyglot authors keep the entire modified copy of AFL in their source tree. Polyglot grammar specifications are given with a mix of JSON specification, Python code, and other formats that are specific to Polyglot.

5.2 Fuzzing Dafny

In the summer of 2021, one of the authors of this paper[4] began the XDsmith project [12] using Xsmith to fuzz Dafny [14], a verification-aware programming language. He was experienced with Racket but had no previous experience with Xsmith. In about a week, he prototyped his Xsmith-based Dafny generator. During a three-month period, he improved his fuzzer and found 28 Dafny bugs. Since that period, his fuzzer has found an additional 2 bugs in the open source Dafny implementation. His fuzzer implementation has 1,666 lines of code, as well as differential testing and verification testing code totaling 722 more lines.

While Dafny fuzzing primarily used differential testing (comparing different Dafny compiler back ends) and compiler error detection, it also included a verification oracle that found one bug. Additionally, one bug was found as a side-effect of writing the fuzzer. While the author was trying to determine the proper type constraint to write for one feature of the fuzzer, he decided to manually test violations of the constraint, and found a bug.

This experience shows that Xsmith can be utilized to write an effective fuzzer in a small time period with a small amount of code.

5.3 Summary of Bugs Found

We have found bugs using Xsmith fuzzers for various programming languages, listed in Figure 6. All reported bugs are unique.

Aside from one Racket bug that had been fixed before we found it, all bugs listed are new (not publicly reported or fixed before we discovered them through fuzzing). All Racket bugs were confirmed by Racket's maintainers, and all but one have been fixed. All Dafny bugs and issues were confirmed by Dafny maintainers, and 6 have been fixed. All of the WebAssembly and Standard ML bugs we found have been confirmed, and most of them have been fixed.

We have written fuzzers for various other languages besides those in the bug table, such as Python, JavaScript, and Lua. However, we have not performed extensive fuzzing with them or with the WebAssembly or Standard ML fuzzers. These less-used fuzzers likely all need at least minor improvements to be effective at finding bugs.

Approximately half of the bugs found by Xsmith-based fuzzers so far have been semantic errors detected by differential testing. These bugs can effectively only be found by program generators that reliably generate conforming test cases. Otherwise, if semantically valid but non-conforming (or semantically invalid) programs are regularly generated, differential testing oracles would be overrun with false positive results, and would be practically useless.

Similarly, approximately half of the bugs found could only effectively be found by generators of semantically correct (though not necessarily conforming) program generators. Bugs characterized by valid programs failing to compile would have too many false positives if tested using a generator that does not reliably generate semantically valid test cases. Some bugs found were characterized by a "successful" compilation that produced ill-formed output. Testing for ill-formed output when the compiler is successful could reasonably be performed with generators of semantically or even syntactically invalid code, but such bugs would likely be difficult for such generators to trigger.

In our experiments, Xsmith program generators have found few crash bugs, the bug class most commonly found by fuzz testing. Differential fuzz testing with Xsmith appears to be very complementary to other fuzzing practices, such as fuzzing with generators of random bytes such as AFL [25]. Xsmith seems well suited to finding a different class of bugs than tools like AFL, and Xsmith does not effectively stress early compiler stages such as parsing.

5.4 Bug Discussion

We present and discuss a small selection of bugs found. The code snippets presented are simplified presentations to illustrate the bugs, not the actual code generated by Xsmith.

5.4.1 Racket and Chez Scheme Float Modulo Bug.

When using a large floating point number, Racket CS (Racket built on Chez Scheme) would give wrong answers to the modulo operator. The bug was found through differential testing.[5]

```
#lang racket/base
;; This number is big enough that despite
;; the .1 it passes `integer?`.
(define num
  ;; The number has been shortened to fit inside
  ;; margins.  The real number had 14 more digits.
  93674811510424315205562331463211094477254417232.1)
(println (integer? num)) ;; Prints true
;; But modulo doesn't stay in bounds of the divisor.
(println (modulo num 10))  ;; Prints a number >10
```

It was determined that it was actually a bug in Chez Scheme itself, and was fixed.[6]

5.4.2 Racket BC GCD Bug.

This is an example of a bignum boundary bug. The bug was determined to be at least 20 years old, and included in the oldest repository import to CVS. The bug was found by differential testing.[7]

[4]The XDsmith author was invited to become a co-author of this paper after writing the fuzzer. At the time of writing the fuzzer, he was completely independent of Xsmith's original authors.

[5]This bug was submitted as https://github.com/racket/racket/issues/3469.

[6]The Chez Scheme bug was fixed in https://github.com/cisco/ChezScheme/pull/537.

[7]Reported as https://github.com/racket/racket/issues/3484.

Language	**Implementation**	**Crash Bugs**	**Semantic Bugs**	**Static Non-crash Bugs**	**Total**
Racket	BC	0	5	0	5
Racket	CS	0	4	0	4
Racket	Both Back Ends	0	2	0	2
Total Racket Bugs					**11**
Dafny	Java Back End	0	0	8	8
Dafny	C# Back End	1	3	3	7
Dafny	Go Back End	0	3	0	3
Dafny	JavaScript Back End	0	4	1	5
Dafny	All Back Ends	0	1	7	8
Total Dafny Bugs					**30**
WebAssembly	Wasmer	1	4	0	5
Standard ML	ML MLKit	0	0	1	1

Crash Bugs: Bugs characterized by a memory error or assertion violation in the compiler or interpreter causing the system under test to exit abnormally. This is not simply failure to compile an input or an interpreter exiting with an exception.
Semantic Bugs: Bugs characterized by a wrong program result. Found primarily by differential testing, but also by testing various properties. For example, a raised exception when the fuzzer has been constrained not to generate code that raises exceptions.
Static Non-crash Bugs: This category includes various kinds of bugs whose finding would have required syntactically and semantically correct programs but not necessarily conforming programs. For example, this category includes failure to compile correct programs, ill-formed compiler output (eg. Dafny compiler outputting ill-formed Java code that the Java compiler can't compile), etc.

Figure 6: Bugs Found with Xsmith-Based Fuzzers

```
#lang racket/base
(define num -4611686018427387904)
;; The gcd function should always return non-negative
;; numbers, but RacketBC returns a negative number.
(gcd num num)
```

5.4.3 Racket Serialization Bug. Besides differential testing, some bugs are found with various properties. For example, our Racket fuzzer was designed to produce code that does not raise exceptions. Thus, a program that raises an exception is evidence of a bug (in Racket or in our fuzzer). This bug was present in both Racket BC (the old C back end for Racket) and Racket CS (the new Chez Scheme back end), and was found with an interesting property.

The `write` and `read` functions are primitive serialization and deserialization functions in Racket and Scheme. In version 7.9, the latest release at the time of fuzzing, the handling in the `read` function for Unicode character U+FEFF, the byte-order-mark, was changed. However, the `write` function was not changed. Thus, data could be altered in a round-trip between the `read` and `write` functions.

When printing generated Racket programs, our Racket fuzzer pretty prints them using a function that ultimately uses the `write` function. When compiling our generated programs, the Racket compiler uses the `read` function. The bug was found because the compiler was rejecting ill-formed programs that were mangled between generation and compilation by the `write` and `read` mismatch.[8] We found multiple `write` and `read` mismatch bugs in this manner.

5.4.4 Dafny Bugs Related to Zero Multiplicity. This class of bugs was found with differential testing. Internally, the multiset data structure in Dafny is implemented as a dictionary mapping from elements to the multiplicity. Many multiset operations assumed the invariant that the multiplicities will always be positive. However, this invariant in fact did not hold, as the multiplicity changing operation can break the invariant. The following code would produce `true false` when compiled to C#, `false true` when compiled to Go, `true true` when compiled to JavaScript, and `false false` (which is the correct output) when compiled to Java.[9]

```
method Main ()
{
  var a := multiset{12}[12 := 0];
  var b := multiset{42};
  print 12 in a, " ", a == b, "\n";
```

[8]This bug was reported as https://github.com/racket/racket/issues/3486.
[9]These bugs were reported as https://github.com/dafny-lang/dafny/issues/1359 and https://github.com/dafny-lang/dafny/issues/1361.

```
}
```

5.4.5 Dafny Bug Found by Verification Testing.

In addition to differential testing, generated Dafny programs were also used for verification testing, which aims to find soundness and precision issues in the Dafny verifier. Additionally, it is useful for finding discrepancies of the underlying semantics between the verifier and the compilers. Given a generated Dafny program with `print` statements, verification testing compiles and runs the program, and correlates `print` statements with their outputs. The program is subsequently transformed to turn the `print` statements into assertions. In the most basic form of verification testing, we expect that these assertions should be verified by the verifier. These assertions therefore test the Dafny verifier's ability to predict the output emitted by the compiled program.

In the following bug, all compiled Dafny programs returned the same incorrect answer, so the bug could not be discovered by differential testing. Yet, it was determined to be incorrect by the verification testing. The problem was that the superset operation was compiled into a subset operation.[10]

```
method Main ()
{
  var a := {1};
  var b := {1, 2};
  print a > b;
}
```

6 Discussion

We discuss a qualitative comparison of Xsmith to other systems for creating programming language fuzzers, and discuss Xsmith's limitations.

6.1 Comparison to Polyglot

Polyglot [4] is a generic language processor that can be configured to produce programs in different languages by providing configuration in a custom format. Polyglot has been very effective at finding crash bugs, finding over 100 bugs in implementations of 9 programming languages. Polyglot uses constrained mutation and a semantic validation step that improves its probability of generating semantically valid programs. These features prevent Polyglot from generating programs with problems such as references to undefined variables, and prevents many type errors. However, these steps do not guarantee semantic correctness. In their evaluation of Polyglot, Chen et al. show that none of their generators produces semantically valid test cases more than 60% of the time. This rate of producing semantically invalid test cases renders it ineffective as a generator for oracles that consistently require semantically valid test cases to prevent false positives, including differential testing.

6.2 Comparison to StarSmith

The StarSmith [13] program generator is a generic framework for generating semantically correct programs. It is configured using a DSL called LaLa, in which users specify the grammar and other rules for program generation, similar to Xsmith. While StarSmith has no built-in or default steps to prevent undefined or other behaviors that are unsuitable for differential testing, users may write custom LaLa code to prevent some such behaviors. For example, the StarSmith authors created a Lua fuzzer that includes a filter preventing the generation of any events in positions where the order of operations is not guaranteed.

While LaLa makes it possible to create a fuzzer for differential testing, it does not encapsulate common patterns to make it easy. If StarSmith users want to make a similar fuzzer that prevents unspecified effect ordering, they would need to write a similar filter. Additionally, this filter is stricter than Xsmith's generic effect analysis, which can still allow non-conflicting effects when evaluation order is unspecified.

Aside from this observation about encapsulating patterns, it is difficult to compare the suitability of Xsmith and StarSmith for differential testing, since StarSmith's authors expressed a lack of confidence that their programs were actually free of undefined behavior. They reported total instances of test cases uncovering a bug, from fuzzing with conforming configurations, rather than unique bugs found or confirmed as they did for other generators. This makes it difficult to know if they found many bugs or few bugs repeated many times. However, they did report unique bugs for their differential testing of SQL, in which they found 11 semantic bugs and 13 segfault bugs among 3 SQL implementations using a SQL generator of approximately 3500 lines of code in their LaLa DSL. This generator was significantly larger than their other generators. Creating specifications for StarSmith to create conforming program generators is possible, but its focus appears to be on program generators for crash fuzzing.

6.3 Limitations of Xsmith

While Xsmith inhabits a new and useful point in the space of fuzzing tools, it has various limitations.

6.3.1 Type System.

Xsmith's type system is flexible but not comprehensive. For example, Xsmith can not currently generate functions with parameters that can be multiple different types but that are not fully generic enough to allow any type, such as a function that accepts either a string or an integer but no other type. Additionally Xsmith can not generate functions with optional arguments or other forms of variadic functions. Built-in functions with such types can be specified as productions in an Xsmith grammar, allowing

[10]This bug was reported as https://github.com/dafny-lang/dafny/issues/1357. Because the output of the `print` statement is incorrect, when it is turned into an assertion the verifier detects that the assertion is violated, thus revealing the bug.

Xsmith to generate calls to them. However, for complicated types, multiple grammar nodes may need to be specified to allow x generation of uses of all potential types of a function.

The type system also lacks a notion of classes. While we have implemented structural records with subtyping and nominal records, which can be used to represent some aspects of object-oriented languages, we have not yet designed a generic system to encode the semantics of classes and objects that is well-suited to a variety of languages. Because classes, objects, and methods have widely varying semantics in different languages, it is difficult to determine what essential features such an encoding should have.

Another limitation of the type system currently is a lack of "negative types" to constrain type variables. There are often situations where one or more particular types are disallowed but where any other type is allowed. An example situation is a position where functions are disallowed, but base types like `int` and `string` are allowed, as well as composite types such as `list` composed of other allowed types (perhaps recursively). Rather than specifying that function types are disallowed, users must enumerate allowed types. Since the set of allowed types may be large or infinite, users might only enumerate a subset of allowed types.

6.3.2 Probabilities. Xsmith follows a long history of using weight specifications to determine the probability of generating any given grammar production[22]. Besides weights, the probability of generating any given production is also affected by filters that consider the type system, the effect system, AST depth, and other factors. While Xsmith's weighting system is flexible and allows for dynamic weight determination, it is difficult to determine how any weighting scheme will ultimately affect the probability of generating a particular production or combination of productions.

Xsmith provides a logging mechanism to view the frequency at which different productions were generated or under consideration for generation. However, logging does not provide a guide to understand how to improve a weighting scheme to achieve a more desired probability distribution, nor does it provide insight into what distributions would be effective at finding bugs.

6.3.3 Effects. The generic effect analysis in Xsmith is both conservative and limited. Because it is conservative, limiting generation to only programs that are guaranteed to be free of unspecified effect ordering, Xsmith filters out many choices that would lead to generating valid, conforming programs. This limitation particularly affects higher-order values. For example, Xsmith has no analysis to determine whether two container values (such as arrays) are the same, so it conservatively assumes that any mutation to a particular container type may conflict with any other access or mutation of that same container type. Because the effect analysis must be generic enough to analyze programs in many languages with varying semantics (most of which is unspecified within an Xsmith fuzzer), the analysis is very limited. Therefore the set of rejected programs is very large, and has great potential to include many interesting and bug-inducing programs.

Xsmith could potentially include a more precise effect analysis by including a way for users to specify language-specific value- and control-flow analyses. However, that would greatly increase the difficulty of writing a new fuzzer, and it is unclear how much it would improve a fuzzer's bug-finding capabilities.

7 Related Work

We compare Xsmith to related systems. In particular, we discuss conforming program generators, program generator generators, and grammar-directed fuzzers.

7.1 Conforming Program Generators

Some random testers for programming languages generate conforming programs, or programs that are syntactically and semantically valid as well as being free from nondeterminism, undefined behavior, and unspecified behavior. An early major conforming program generator was Csmith [24]. Similar systems include Verismith [10] and SQLsmith [21]. YARPGen [15] statically generates programs that are free from undefined behavior with no dynamic checks. Csmith and other conforming program generators have been very successful at finding bugs in programming language implementations. These bugs include semantic bugs found with differential testing that can not be found by more common crash oracles. However, conforming program generators like Csmith require much programmer time and tend to be tens of thousands of lines of code to implement.

JFQ [19] is a framework for imperative property-based testing that has been successfully used to fuzz programming languages. JFQ allows users to write correctness properties for generated data. However, JFQ does not include a general framework for program analysis to generate conforming programs, so users must write their own analyses.

Xsmith is a DSL and library for building conforming program generators like Csmith, but with less time and code. Xsmith eases development of conforming program generators by providing a declarative DSL, along with generic analyses, generation strategies, and other tools as a library. Thus Xsmith allows users to build conforming program generators at a fraction of the cost of stand-alone generators like Csmith.

7.2 Program Generator Generators

Polyglot [4] is a framework that takes a language specification in a custom format and produces random programs. Polyglot includes a semantic validation step that improves the percentage of syntactically and semantically valid test cases generated. However, programs generated by Polyglot are not guaranteed to be syntactically or semantically correct, or to be conforming. Polyglot has been successfully used to find well over 100 bugs in implementations of at least 9 programming languages. However, it is not a suitable system for finding semantic bugs via differential testing because generating non-conforming programs leads to impractically high false positive rates.

StarSmith [13] is a framework that takes a language specification in a DSL called LaLa and produces random programs. StarSmith generates programs that are syntactically valid and well-typed. Some StarSmith generators have been further crafted to generate programs that are free of unspecified behavior for differential testing. StarSmith generators that have filters to generate only conforming programs must individually be constrained to not generate offending code. For example, the StarSmith authors created a Lua generator that included custom code to prevent any effects when the order of evaluation is not guaranteed. Such code would need to be repeated for each generator in StarSmith for languages without guarantees of evaluation order.

PLT Redex [6] is a framework that allows users to specify a semantics for a programming language. It includes a testing feature that generates random well-typed programs [7]. However, it does not generate programs that are free from undefined behavior. This is useful in Redex, since it helps users to find where undefined behavior exists in their semantics definitions, however it limits its use as a generator for differential testing.

LangFuzz [11] is a fuzzer that uses grammar-directed fuzzing. Additionally, LangFuzz can incorporate code patterns learned by parsing program corpuses. LangFuzz has successfully found many bugs. However, LangFuzz does not generate conforming programs that are free from undefined behavior, limiting its practical utility for differential testing.

Xsmith has been designed to generate programs suitable for differential testing. Xsmith fuzzers generate programs that are syntactically and semantically valid, in addition to being free from undefined or nondeterministic behavior. Xsmith includes generic effect analyses for the common case of unspecified order of evaluation, allowing programs to include effects while still preserving a well-defined relationship between effects. While some unspecified behaviors need to be handled on a per-language basis, Xsmith's canned components library helps with some common patterns.

7.3 Grammar-Directed Fuzzers

Some fuzzers, such as Lava[22] and Yagg [5], are grammar-directed[1, 8, 9, 16], meaning they only build program trees that match a given grammar. These fuzzers are useful because they can generate syntactically valid test cases that pass early stages of a compiler or other language processor, allowing fuzzing to exercise deeper code paths. However, grammar-directed fuzzers, without other guidance, do not guarantee that their outputs semantically correct or conforming. Thus they can not reliably be used to find semantic bugs through differential testing.

Some grammar-directed fuzzers such as Grimoire [2] automatically learn a grammar instead of taking a user-supplied grammar is input. While this automatic learning step means that the fuzzer requires less up-front effort to craft a grammar specification, it provides even weaker guarantees about the syntactic and semantic validity of produced outputs.

Xsmith is grammar-directed in addition to being directed by other analyses, such as type and effect analyses. These analyses constrain program generation to produce conforming programs that are useful for finding semantic bugs.

8 Conclusion

Differential fuzz testing can be a powerful tool for finding semantic bugs in programming languages. Xsmith is a library and DSL that provides shared infrastructure, declarative specification, and extension hooks that allow users to easily build featureful fuzz testers for differential testing. Xsmith has been used to create a variety of fuzzers requiring modest effort and code size that have found bugs in different language implementations. Compared to related work, Xsmith is the first tool focused on easily creating fuzzers for differential testing. Xsmith fuzzers can be used synergistically with other fuzzing techniques to find bugs in many language implementations.

Acknowledgments

We thank the anonymous GPCE reviewers for their valuable comments on this work. We performed our experiments in the Utah Emulab [23] testbed. This material is based upon work supported by the National Science Foundation under Grant Number 1527638.

References

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

[2] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure While Fuzzing. In *Proc. 28th USENIX Security Symposium*. 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[3] Christoff Bürger. 2015. Reference Attribute Grammar Controlled Graph Rewriting: Motivation and Overview. In *Proc. 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. 89–100. https://doi.org/10.1145/2814251.2814257

[4] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proc. 42nd IEEE Symposium on Security and Privacy (S&P)*. 642–658. https://doi.org/10.1109/SP40001.2021.00071

[5] David Coppit and Jiexin Lian. 2005. Yagg: An Easy-to-Use Generator for Structured Test Inputs. In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–359. https://doi.org/10.1145/1101908.1101969

[6] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press.

[7] Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, 383–405. https://doi.org/10.1007/978-3-662-46669-8_16

[8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 206–215. https://doi.org/10.1145/1375581.1375607

[9] K. V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (Dec. 1970), 242–257. https://doi.org/10.1147/sj.94.0242

[10] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proc. 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA).* 277–287. https://doi.org/10.1145/3373087.3375310

[11] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. 21st USENIX Security Symposium.* 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[12] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (Experience Paper). In *Proc. 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 556–567. https://doi.org/10.1145/3533767.3534382

[13] Patrick Kreutzer, Stefan Kraus, and Michael Philippsen. 2020. Language-Agnostic Generation of Compilable Test Programs. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (ICST).* 39–50. https://doi.org/10.1109/ICST46399.2020.00015

[14] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[15] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

[16] Peter M. Maurer. 1990. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

[17] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[18] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[19] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 398–401. https://doi.org/10.1145/3293882.3339002

[20] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 329–340. https://doi.org/10.1145/3293882.3330576

[21] Andreas Seltenreich. 2020. SQLsmith software repository. https://github.com/anse1/sqlsmith

[22] Emin Gün Sirer and Brian N. Bershad. 1999. Using Production Grammars in Software Testing. In *Proc. 2nd Conference on Domain Specific Languages (DSL).* 1–13. https://www.usenix.org/conference/dsl-99/using-production-grammars-software-testing

[23] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI).* 255–270. https://www.usenix.org/legacy/event/osdi02/tech/white.html

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 283–294. https://doi.org/10.1145/1993498.1993532

[25] Michał Zalewski. 2020. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/

Received 2023-07-14; accepted 2023-09-03