This is a preprint of a paper that appeared in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 2020, pp. 39-50, DOI: 10.1109/ICST46399.2020.00015.

Language-Agnostic Generation of Compilable Test Programs

Patrick Kreutzer, Stefan Kraus, Michael Philippsen
Programming Systems Group, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
patrick.kreutzer@fau.de, stefan.kraus@fau.de, michael.philippsen@fau.de

***Abstract*—Testing is an integral part of the development of compilers and other language processors. To automatically create large sets of test programs, *random program generators*, or *fuzzers*, have emerged. Unfortunately, existing approaches are either language-specific (and thus require a rewrite for each language) or may generate programs that violate rules of the respective programming language (which limits their usefulness).**

This work introduces ∗Smith, a *language-agnostic* framework for the generation of *valid*, *compilable* test programs. It takes as input an abstract attribute grammar that specifies the syntactic and semantic rules of a programming language. It then creates test programs that satisfy all these rules. By aggressively pruning the search space and keeping the construction as local as possible, ∗Smith can generate huge, complex test programs in short time.

We present four case studies covering four real-world programming languages (C, Lua, SQL, and SMT-LIB 2) to show that ∗Smith is both efficient and effective, while being flexible enough to support programming languages that differ considerably. We found bugs in all four case studies. For example, ∗Smith detected 165 different crashes in older versions of *GCC* and *LLVM* (50 more than Csmith, a state-of-the-art fuzzer for C).

∗Smith and the language grammars are available online under the terms of an open-source license.

***Index Terms*—fuzz testing, compilers, attribute grammars**

I. Introduction

Since compilers (and other kinds of language processors) are a central component of our software-driven world, their correctness is crucial: If a compiler is erroneous, it fails to translate applications or even silently produces wrong code. Unfortunately, real compilers *do* contain bugs, as is evidenced by the massive amount of research studies (cf. Sec. II) and entries in bug trackers. Since a formal verification of these highly complex software systems is often impractical, testing is the only remedy to build confidence in their implementation.

Fuzzing is a well-established technique to perform this testing process in an automated way. It consists of two basic steps: *1)* The compiler under test is fed with randomly generated test programs. *2)* The compiler's behavior and/or output is examined to check if a test program triggers a bug.

Depending on the testing scenario and goals, there are several options for step 2 (see Chen et al. [1] for a comparative study). Basic testing runs the compiler on a test program. If the compiler crashes (e.g., due to a failed assertion), there is a bug. *Randomized Differential Testing* (*RDT*) [2] feeds the same test program into different compilers and compares the outputs of the generated executables. *Equivalence Modulo Inputs* (*EMI*) [3] takes a test program and an input I. It then generates program variants that should produce the same output for I as the original program (e.g., by pruning dead code). To find bugs in a compiler, *EMI* compiles all variants and compares the outputs of the generated programs.

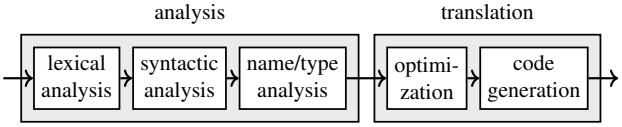


Fig. 1: Stages of an exemplified compiler.

Key to all these approaches is the generation of complex test programs that cover as many parts of the compiler as possible. What makes step 1 complicated is the way compilers typically work, see Fig. 1. If an analysis stage detects an error in the input program, the compiler terminates with an error message. Later, particularly error-prone stages are not executed and thus not tested. This also restricts techniques like RDT or EMI.

There are approaches that try to solve this problem. On the one hand, there are *language-specific* fuzzers that generate programs in a particular programming language. For example, Csmith [4] generates C programs that are not only compilable but also free of undefined behavior. In combination with *RDT*, Csmith was shown to be a highly effective bug finding tool for C compilers. One drawback of language-specific approaches is that the syntactic and semantic rules of the respective programming language are entangled with the (non-trivial) program generation logic. This not only hampers maintainability and extensibility, but also requires a rewrite from scratch to support a different programming language [4].

On the other hand, there are *language-agnostic* approaches that can generate programs in a variety of languages. For example, there are generators [5]–[7] that take as input a context-free grammar of a programming language and generate strings that conform to this grammar. Users only have to focus on writing an appropriate grammar. But while the generated programs are guaranteed to be syntactically correct, they most likely do not pass the name/type analysis. Thus, these programs can only be used for a shallow testing of compilers. To the best of our knowledge, there is no language-agnostic program generator yet that produces *compilable* programs for *arbitrary*, *real* programming languages and that scales to the generation of *many*, *large test cases*. ∗Smith fills this gap.

Fig. 2 depicts the workflow of ∗Smith. In our newly designed language LaLa (*language specification language*),

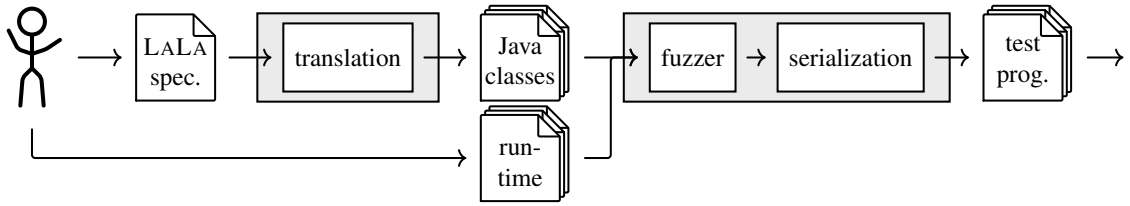


Fig. 2: Workflow of ∗Smith, our *language-agnostic* fuzzing framework that generates *compilable* test programs.

the user defines the syntactic and semantic rules of a programming language in a declarative way by means of an attribute grammar (see Sec. III). This *specification* may reference methods from *runtime libraries* written in Java that the user also provides. ∗Smith *translates* a given specification to Java classes that, together with the runtime libraries, are the input for the *fuzzer*. It creates arbitrarily many abstract syntax trees (ASTs) of programs that conform to the specified rules. The algorithm exploits novel technical ideas that make it efficient (see Sec. IV). For example, we introduce *fail patterns* that allow us to aggressively prune the search space. A final *serialization* step converts the ASTs to *test programs*.

Sec. V presents four case studies covering four different languages (*C*, *Lua*, *SQL*, and *SMT-LIB 2*) to show that ∗Smith is *flexible* (it supports different languages with diverse properties), *efficient* (it can create huge programs in a short time), and *effective* (it detects bugs in production compilers).

Our implementation and example specifications are available online (`https://github.com/FAU-Inf2/StarSmith`).

II. Related Work

Compilers have been tested with random programs before. We discuss important work and how ∗Smith differs. See Liang et al. [8] for a recent overview of general fuzzing techniques.

There are many language-specific compiler fuzzers, e.g., *Csmith* [4] and others [2], [9]–[12] for the C programming language, *CLsmith* [13] for OpenCL, *GLFuzz* [14] for OpenGL, *jsfunfuzz* [15] for JavaScript, or the tool by Yoshikawa et al. [16] for Java bytecode. The number of publications not only shows that there is a clear need for compiler fuzzers, but also for a language-agnostic approach that reduces the effort required for developing a new fuzzer.

There are some language-agnostic test program generators, but they do not offer ∗Smith's level of conformity or flexibility. For example, some of them cannot guarantee that the generated programs are semantically (or even syntactically) valid. *LangFuzz* [17] and *IFuzzer* [18] take as input a corpus of example programs as well as a context-free grammar to parse these. Based on the ideas of evolutionary algorithms, they apply different mutation operations to the parse trees to generate new programs. The resulting test programs are valuable in case of dynamically typed languages (e.g., JavaScript, PHP), but for statically typed languages the programs often do not compile. After training a deep neural network on human written example code, *DeepSmith* [19] emits new test programs. Compared to ∗Smith, this means less work for users, but the *DeepSmith* output is not guaranteed to be compilable.

The *lava* system [20] generates test programs that conform to a context-free grammar, augmented with *actions* and *guards* to check simple contextual rules. Its expressiveness and code generation are sufficient for low-level languages like Java bytecode, but *lava* does not provide a solution for higher-level, structured languages with a complex type system.

Given a code fragment and a set of variables as input, *skeletal program enumeration* replaces the fragment's identifiers with all possible combinations of variable usages. Zhang et al. [21] propose an efficient, language-agnostic solution, but it cannot generate new test programs from scratch.

Pałka et al. [22] generate random lambda terms based on the simply-typed lambda calculus. These terms can be translated to type-correct λ-expressions in different programming languages. ∗Smith is more flexible and does not require a formal definition of the type system (which often is not available).

Besides the detection of compilation bugs, there are other uses of compiler fuzzing, e.g., to discover missed optimizations [23], to reveal faulty or missing warning messages [24], or to test other language processing tools, e.g., static analyzers [25], symbolic execution frameworks [26], or refactoring engines [27]. ∗Smith could reduce the effort to adapt such approaches to other languages.

To increase its usefulness in practice, compiler fuzzing is often accompanied by other methods, in particular concerning test case *reduction* [28]–[32] and *prioritization* [33]–[35]. ∗Smith could be extended with such features in the future.

Finally, to eliminate the need for testing, research has introduced *formally verified* compilers, e.g., the CompCert C compiler [36] or the CakeML compiler [37]. But formal verification requires a lot of work and bugs can still be found for a variety of reasons [4], [21], which is why, in general, formally verified compilers are not (yet?) popular in practice.

III. The LaLa Specification Language

This section briefly introduces the LaLa specification language and establishes the terminology that we use below. At its core, LaLa allows the specification of an *attribute grammar* [38], [39] that describes syntactically and semantically valid programs represented as abstract syntax trees (ASTs).

Our running example is a simple language that solely allows assignments of expressions to variables. Expressions may only use variables that have been defined before. Checking this rule requires *context* information that cannot be provided by a simple context-free grammar. (Of course, LaLa can describe much more complex rules, see Sec. V.)

```
use SymbTab;

class Program {
  prog("${stmts : StmtList}\n") {
    stmts.symbs = (SymbTab:empty);
  }
}
@list
class StmtList {
  inh symbs : SymbTab;
  one("${s: Stmt}") {
    stmt.s_before = this.symbs;
  }
  @weight(3)
  mult("${s: Stmt}\n${r : StmtList}") {
    s.s_before = this.symbs;
    r.symbs = s.s_after;
  }
}
```

```
class Stmt {
  inh s_before : SymbTab;
  syn s_after : SymbTab;
  assign("${i : Identifier} := ${e : Expr};") {
    e.symbs = this.s_before;
    this.s_after = (SymbTab:put this.s_before i.str);
  }
}
class Expr {
  inh symbs : SymbTab;
  num("${val : Number}") {}
  use_var("${var : UseVariable}") {
    var.symbs = this.symbs;
  }
  binop("(${l : Expr}) ${op : Op} (${r : Expr})") {
    l.symbs = this.symbs;
    r.symbs = this.symbs;
  }
}
```

```
class UseVariable {
  inh symbs : SymbTab;
  grd defined;
  var("${var : Identifier}") {
    this.defined =
      (SymbTab:contains this.symbs var.str);
  }
}

# alternatively:
class UseVariableGen {
  inh symbs : SymbTab;
  var(SymbTab:defs this.symbs) : String {}
}

class Identifier("[a-z]{1,3}");
class Number("0|[1-9][0-9]{0,2}");
class Op("+|-|*|/");
```

Fig. 3: LaLa specification for a simple language that allows assignments of expressions to variables.

A. Basic Concepts and Terminology

Fig. 3 holds the LaLa specification for the example language. Initial `use` statements declare the runtime classes to be imported (e.g., `SymbTab` for a symbol table implementation). Then there are a number of *classes* that express the *terminals* and *non-terminals* of classic (attribute) grammars and usually correspond to the syntactic structures of the programming language (e.g., `Stmt` for statements or `Expr` for expressions). Each AST node belongs to exactly one class.

Each class declares one or more *productions* that represent the alternative ways to instantiate a node of this class (e.g., `num` in class `Expr` for a single number or `binop` for binary expressions). A production not just specifies the node's AST children but also its textual serialization. For example, `StmtList` nodes instantiated with the `mult` production have two children — `s` of class `Stmt` and `r` of class `StmtList` — and are serialized by recursively serializing `s` and `r`, with a newline (`\n`) between them. A serialiazition may also contain calls of runtime methods that are evaluated once in the final serialization step and that can be used to embed additional code in the fuzzer's output (e.g., code that prints the values of all defined variables).

Additionally, LaLa offers *literal classes* that specify their possible instantiations by means of a regular expression. In the example language, `Identifier` nodes are random strings of one to three characters in the range from `a` to `z`.

Classes and productions suffice to specify the context-free parts of a programming language, i.e., its syntactic rules. To propagate values along the AST and to specify the context-sensitive rules (e.g., that a variable has to be defined before it can be used in expressions), classes declare *attributes*. A node that belongs to class C carries values for each of C's declared attributes. Nodes compute their own *synthesized attribute values* (`syn`) and pass them bottom-up to their respective parents. Literal classes define an implicit, synthesized `str` attribute that holds the string representation. Parent nodes compute the *inherited attribute values* (`inh`) of their children top-down. To specify contextual rules, some classes declare *guard attributes* (`grd`), i.e., boolean conditions that programs have to satisfy.

An *evaluation rule* in the body of a production specifies how a *target* attribute is computed from a set of *source* attributes. There must be evaluation rules for all synthesized attributes of the containing class as well as for all inherited attributes of all children. In general, these rules call methods of the runtime classes that implement the logic. In the example, the `var` production of the class `UseVariable` passes both the current symbol table and the name of a variable to the method `contains` of the `SymbTab` runtime class to check if the variable has already been defined. Only if this guard holds, the sub-tree rooted at the `UseVariable` node is valid in the given context.

For cases like these, in which the given context dictates a set of possible atomic values, LaLa also offers *generator classes*. These contain a call of a runtime method that — depending on the given context information — returns a list of all valid values. Fig. 3 shows this with the alternative `UseVariableGen` class which uses the `defs` runtime method to determine the list of all defined variables. Generators reduce the overall runtime of our fuzzer: With `UseVariable`, our algorithm has to repeatedly choose and discard random variable names until it finds a valid one (see Sec. IV). On the other hand, using the generator class `UseVariableGen`, our algorithm can simply pick one of the generated values at random. Generator nodes are implicitly guarded (such a node is valid in the given context if the list is non-empty, i.e., if there is at least one valid value). The only difference to normal classes is that the productions of a generator node are created *dynamically* once the construction reaches the node (i.e., one production per value).

Finally, LaLa provides several *annotations* for classes and productions (e.g., `@list` or `@weight`) that influence the decisions made by the fuzzing algorithm (see Sec. IV).

B. Imposed Requirements

If attribute A is a (direct or indirect) source value for B, B *depends* on A. If A and B are mutually dependent, computing their values is impossible in general. We exclude this case by requiring that the grammar is strongly non-cyclic [39]. In practice, this does *not* severely limit the languages that can be defined [39]. Attribute values may still flow in arbitrary directions through the AST and nodes may still mutually depend on each other (by using disjoint attribute sets).

Furthermore, for our fuzzing algorithm to work, we assume additional, implicit dependencies and require that the grammar

is still strongly non-cyclic: For a guarded class, all its synthesized attributes implicitly depend on all its guard attributes, which in turn implicitly depend on all its inherited attributes. This excludes the case that two guarded nodes are mutually dependent. Thus, *any* set of guarded nodes is *topologically sortable*, i.e., it must have at least one node that does not depend on the others. This allows our fuzzing algorithm to always choose where to proceed next, see Sec. IV for details. This additional requirement also does *not* severely limit the expressiveness of LaLa. It is still possible to describe real-world programming languages with complex rules, see Sec. V.

IV. Fuzzing Algorithm

The goal of our fuzzing algorithm is to efficiently generate complex test programs that conform to a LaLa specification. Two observations make this task difficult: *1)* The search space induced by typical specifications is huge and the vast majority of possible program trees violate guard conditions. Thus, a program tree generated fully at random is highly unlikely to be valid. *2)* The runtime methods used by the attribute evaluation rules are black boxes; the algorithm first has to generate a concrete (sub-)tree before it can check its validity.

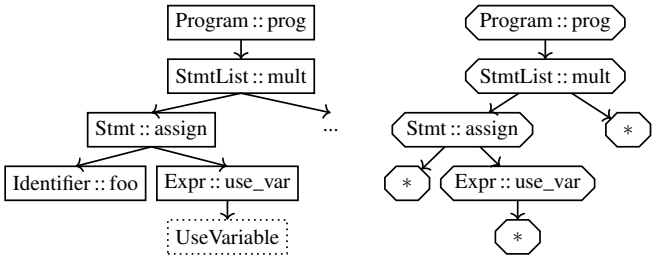
In preliminary experiments we found that a simple backtracking does not suffice, since it gets stuck in sub-trees quite often. We thus propose a more sophisticated algorithm. In a nutshell, it generates a program tree by instantiating its nodes with random productions in a top-down manner. When the evaluation of a guard attribute fails, our algorithm deconstructs parts of the program tree and then replaces them with alternatives. There are two main ideas that make this construction efficient: *1)* Our algorithm learns from mistakes: Whenever it creates an invalid (sub-)tree, it analyzes the current context to exclude other, inevitably failing trees from further consideration. We introduce so-called *fail patterns* to store this information. *2)* To limit the impact of a mistake on the runtime, our algorithm keeps the AST construction as local as possible, i.e., it splits it into multiple, independent constructions of smaller sub-trees.

For presentation purposes, we first introduce preliminary aspects before we explain the full algorithm.

A. Three Preliminary Aspects

1) Fail patterns: During the construction our algorithm may tentatively create a program tree that violates guards. It then discards parts of the tree and constructs an alternative. To ensure that it does not construct the same sub-tree again and to prune the search space, ∗Smith constructs a *fail pattern* that captures prohibited combinations of productions.

Recall that in the example language variables can only be used after their definition, i.e., the right hand side of the very first definition may never use any variables. Assume now that the construction process reaches the program "`foo := <UseVariable>; [...]`", the AST of which is depicted in Fig. 4(a). (We use rectangular nodes for ASTs.) The fuzzer then determines that the `UseVariable` sub-tree cannot be instantiated without violating guards. This leads to the construction of the fail pattern depicted in Fig. 4(b), where a ∗ represents a wildcard for arbitrary trees. (We use chamfered nodes for fail patterns.) The fail pattern excludes from further consideration any AST that contains a `UseVariable` sub-tree on its first assignment's right hand side. Sec. IV-B4 shows how we can deduce solely from the specification that the concrete variable on the assignment's left hand side is irrelevant. This generalized fail pattern ensures that our fuzzer does not simply try to replace `foo` with a different variable when searching for an alternative, as this would inevitably fail again.



(a) A partially constructed AST. (b) Generated fail pattern.

Fig. 4: Fail pattern for an invalid AST.

2) Recursive productions: Directly or indirectly recursive productions allow ASTs of unbounded sizes (e.g., the `binop` production creates two new `Expr` nodes that could again be instantiated with `binop`, etc.) To avoid an unlimited AST growth and to ensure that the program generation terminates, we limit the heights of sub-trees generated by recursive productions.

For this purpose, we compute min_P for each production P, the height of the smallest possible sub-tree constructed by P. In the example, $min_{\texttt{num}}$ is 1, since all sub-trees rooted at a `num` node have height 1, whereas $min_{\texttt{binop}}$ is 2, since the smallest possible sub-trees rooted at a `binop` node have heights 2 (e.g., an expression over two numbers). ∗Smith computes these values during the translation of a LaLa specification and uses them at program generation time to ensure that an AST has a finite height: Whenever ∗Smith instantiates a node N by a recursive production, it enforces a user-defined height limit max_N for the sub-tree rooted at N (and recursively a limit of $max_N - 1$ for its children). The fuzzer only applies a recursive production P to a node N if $min_P \leq max_N$, i.e., if the smallest possible sub-tree that P can produce fits into the height limit of N. In the example, when the construction reaches a node N with $max_N = 1$, it no longer considers the `binop` production for N (it may choose `num` instead).

Children of `@list` classes (e.g., `StmtList`) are siblings in the AST. Thus, they all receive the same height limit.

3) Attribute evaluation: During the generation of an AST, ∗Smith has to repeatedly check if all its guard attributes are satisfied. In general, these values depend on the attribute values of other AST nodes (which in turn may depend on other values etc.). To compute these values, our algorithm repeatedly applies a *dynamic attribute evaluator* [39] to the current AST. It determines and computes all attributes for which all source values have already been computed until a fixpoint is reached. For example, consider again the AST depicted in Fig. 4(a).

At first, the evaluator applies the (only) evaluation rule of the `prog` node (which does not depend on any source attributes) to compute the initial (empty) symbol table for the `mult` child (its inherited `symbs` attribute). This enables the computation of the `s_before` attribute of the `assign` node by applying the respective evaluation rule of the `mult` production etc.

Our evaluator differs from typical use-cases in two ways: *1)* As the AST often holds unfinished sub-trees, in general, not all attribute values can be computed. However, our evaluator can always compute the values that are needed for a decision in the fuzzing algorithm, see below. *2)* As our fuzzer has to repeatedly compute the attribute values, the evaluator works *incrementally* and can exploit that in most cases the fuzzer only adds new nodes to the AST. Attribute values that have already been computed during a previous run of the evaluator do not need to be re-evaluated (they cannot have changed).

Note that we omit the calls of the evaluator in the descriptions below to ease the presentation and discussion.

B. Outline of the Algorithm

If we were concerned with generating *syntactically* valid programs, our fuzzing algorithm could proceed in a strictly top-down manner: It would choose a random production for the program's root node and handle its children recursively until it reaches leaf nodes. Such a simple scheme suffices in the context-free as case all sub-trees are independent.

This no longer holds in our context-sensitive case in which attribute values are propagated through the AST. A sub-tree A at one position in the AST can rule if another sub-tree B at a different position is valid. Thus, A and B cannot be generated independently (or concurrently): To generate B, we need a valid sub-tree A. And if the context prohibits a valid construction of B, we may have to replace A with an alternative (which may require more changes in a sub-tree C etc.).

Our algorithm is based on these observations and thus distinguishes *unguarded* from *guarded* nodes. As unguarded nodes do not impose any context-sensitive constraints themselves, our algorithm first expands them randomly[1] in a top-down manner, i.e., it first creates a skeletal program tree whose leaves are either terminals or unexpanded guarded nodes (this includes implicitly guarded generator nodes, see Sec. III).

It then tries to expand these unfinished nodes recursively (we will later see that it can always decide where to proceed next). This way, our algorithm splits the AST construction into multiple independent and simpler constructions of smaller sub-trees.[2] If a recursive call cannot find a valid sub-tree in the given context, our algorithm first learns a new fail pattern by analyzing this context to prune the remaining search space. It then generates an alternative context (i.e., it alters the skeleton or a recursively constructed sub-tree so that none of the fail patterns prohibits this and so that it only requires small changes to the AST) and tries the completion again. This process is repeated until the AST is complete.

[1]The selection of a random production has to obey the height limits for recursive productions, see Sec. IV-A. Also note that the selection uses a weighted probability and that the user can annotate a production with `@weight` to increase its probability over other productions with their implicit weight of 1 (e.g., choosing `mult` from Fig. 3 is 3 times as likely as choosing `one`).

[2]Users can even boost this effect by annotating classes with `@unit`. Such classes are handled like guarded classes (the guard is always satisfied).

```
01: function FUZZ(root, fps)
02:   {alt}: while ALTERNATIVE(root, fps) do
03:     while (guarded ← FINDGUARDED(root)) ≠ ∅ do
04:       next ← CHOOSENEXT(guarded)
05:       next_fps ← FILTERANDTRIM(root, next, fps)
06:       res ← FUZZ(next, next_fps)
07:       if res ≠ SUCCESS then
08:         fps←fps ∪ {NEWFAILPATTERN(root, next, fps, res)}
09:         continue {alt}
10:     if ALLGUARDSTRUE(root) then return SUCCESS
11:     else fps ← fps ∪ { NEWFAILPATTERN(root) }
12:   DECONSTRUCT(root)
13:   if failed due to height limit then return HEIGHT
14:   else return FAIL
```

Fig. 5: Novel fuzzing algorithm.

Fig. 5 shows the recursive FUZZ procedure that implements our fuzzing algorithm. We explain it in a top-down way; the four subsections below fill in the details.

FUZZ receives as input the $root$ node of the sub-tree that should be created, as well as a list fps of fail patterns that the new sub-tree has to conform to (the very first call of FUZZ receives an empty list of fail patterns).

Each iteration of the {alt} loop (lines 2–11) generates a new ALTERNATIVE that none of the previously learned fail patterns in fps prohibit (see subsections IV-B1 and IV-B2). As motivated above, ALTERNATIVE only handles the unguarded nodes, i.e., it constructs a skeletal program tree that may still contain unfinished guarded nodes (e.g., it may generate the partial AST from Fig. 4(a)). FUZZ handles the remaining guarded nodes (e.g., the `UseVariable` node) by recursively calling itself for each such node in turn (lines 3–9; see subsection IV-B3 for details). To ensure that the recursive construction of the $next$ node does not create a sub-tree that any of the fail patterns in fps prohibit, FUZZ uses FILTERANDTRIM (line 5) to determine the fail patterns that are relevant for the call (see subsection IV-B1). Whenever the recursive instantiation of a guarded node fails (lines 7–9) or the guards of the current $root$ node are not satisfied[3] (line 11), FUZZ creates a NEWFAILPATTERN (see subsection IV-B4 for details) and moves on to the next alternative.

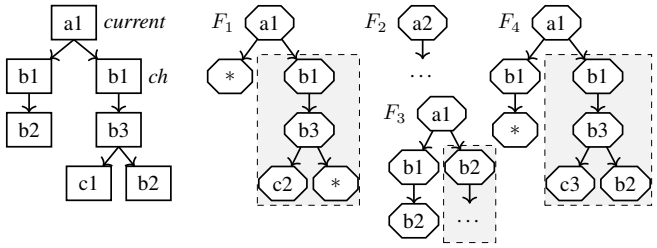
If FUZZ can complete a single alternative to a valid sub-tree that satisfies all guards, height limits, and fail patterns, it returns SUCCESS plus the tree that it built as a side-effect (line 10). If, however, FUZZ cannot find such a tree and there is no ALTERNATIVE left, it deconstructs the last alternative and either returns HEIGHT (if the failure is due to a height limit for recursive productions) or FAIL (lines 12–14).

1) Construction of Alternative Sub-Trees: ALTERNATIVE's task is to construct a skeletal sub-tree that ends with terminals

[3]In some cases, the guards of the $root$ node only depend on some children. Our actual implementation thus checks these guards after *each* recursive call. To ease the presentation, we decided to simplify the algorithm here.

```
01: function ALTERNATIVE(current, fps)
02:     if CONTAINSWILDCARDPATTERN(fps) then return false
03:     if ISGUARDEDCHILD(current) then
04:         DECONSTRUCT(current)
05:         return true
06:     if SHRINKLIST(current, fps) then return true
07:     for all children ch of current in order of dependencies do
08:         ch_fps ← FILTERANDTRIM(current, ch, fps)
09:         if ALTERNATIVE(ch, ch_fps) then return true
10:     return DECONSTRUCTANDREPLACE(current, fps)
```

Fig. 6: Construction of an alternative tree.



(a) Example AST. (b) Four example fail patterns, F_1–F_4.

Fig. 7: Filtering and trimming of fail patterns with respect to the node ch; the dashed boxes depict the results.

or (unfinished) guarded nodes and that none of the previously learned fail patterns prohibit. In theory, each call of ALTERNATIVE could create a new sub-tree from scratch. Since, in general, this would discard previously constructed sub-trees and the work that was necessary to build them, we use a more sophisticated approach that only replaces a small portion of the last alternative (if possible). To do so, ALTERNATIVE is a recursive depth-first traversal on the current sub-tree, see Fig. 6. The traversal either just passes the *current* node (and leaves it unchanged) or replaces the *current* node's sub-tree with an alternative. There are three base cases: *1)* If the fail patterns for *current* contain a wildcard pattern (i.e., a pattern consisting of a single $*$), there is no alternative left for *current* and ALTERNATIVE returns to its caller (line 2). *2)* If *current* is one of the guarded children of the currently constructed *root* node, ALTERNATIVE does not proceed with the traversal, since it only handles skeletal program trees of unguarded nodes. Instead, it deconstructs the *current* sub-tree (see subsection IV-B2) and ends its search for an alternative (lines 3–5). The deconstruction ensures that a future recursive call of FUZZ will construct a new alternative for this sub-tree. *3)* If *current* is the root node of a partially constructed `@list` structure (i.e., if the first n elements of the list have already been constructed successfully, but the construction of the $n{+}1^{st}$ element failed), the list is shrunk to a shorter list containing only the first n elements (unless this is prohibited by any of the fail patterns; line 6). This ensures that our fuzzing algorithm does not try to create infinitely large lists.

In all other cases, *current* is an unguarded node that may or may not have children. If it has, ALTERNATIVE traverses them recursively. If there are multiple children, they are visited in order of their dependencies (line 7), i.e., if ch_1 depends on ch_2 (but not vice versa), ch_1 is visited *before* ch_2. This reduces the overall runtime: If we changed ch_2 first, in general this would also require changes in the sub-tree at ch_1 due to the dependency. Before visiting a child ch, FILTERANDTRIM removes all fail patterns that do not match the context surrounding ch as those fail patterns are irrelevant for the construction of an alternative in ch. Of those whose contexts match, it then trims off this context, i.e., the parts that do not belong to the smaller sub-tree at ch. As an example, consider the AST and fail patterns in Fig. 7. FILTERANDTRIM finds that the fail pattern F_2 is irrelevant for ch since its top-level production $a2$ differs from $a1$ in the AST. The contexts of the other fail patterns (everything outside of the dashed boxes) match the given AST. FILTERANDTRIM thus trims off these contexts and only the dashed boxes are used as fail patterns when generating an ALTERNATIVE for ch.

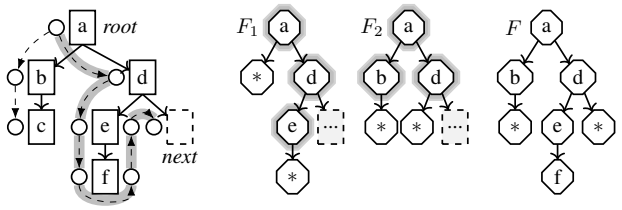
If it cannot find an alternative for any of the children, ALTERNATIVE tries to replace *current* itself (line 10): If there is a skeletal sub-tree that none of the fail patterns in *fps* prohibit, DECONSTRUCTANDREPLACE removes the *current* sub-tree (see subsection IV-B2) and replaces it with a new tree. Otherwise, the search for an alternative returns to the parent.

2) Deconstruction of Sub-Trees: The construction of alternatives may require the deconstruction of previously generated sub-trees. If there were no attributes and guards, we could simply remove the AST nodes. But due to the attributes, a deconstruction may also affect other parts of the AST (think of the example language: removing a variable definition invalidates all its usages elsewhere in the AST). Hence, the deconstruction traverses the AST and deletes all attribute values that can no longer be computed. If the value of a guard attribute is deleted, the respective sub-tree is deconstructed recursively since its validity can no longer be ensured.

This recursive deconstruction is not as costly as it sounds. In practice only few additional sub-trees have to be deconstructed for two reasons: First, since ALTERNATIVE visits the children in order of their dependencies, it in general deconstructs sub-trees that only few other sub-trees depend on. Second, each recursive call of FUZZ constructs a sub-tree independently from the surrounding AST. Thus, a deconstruction can only affect nodes in the currently constructed sub-tree.

3) Instantiation of Guarded Nodes and Generators: As described above, FUZZ is called recursively to extend the guarded nodes of a skeleton (line 6 in Fig. 5). To decide whether a sub-tree for the *next* guarded node satisfies all guards (or to create the productions for generator nodes), in general *all* its inherited attributes need to be known. Luckily, because of our skeleton-first construction, all unguarded nodes have already been constructed and have their attributes evaluated as far as possible. And because of the restrictions on LALA specifications (see Sec. III-B) there is always at least one *next* node that does not depend on other guarded nodes in the current sub-tree. Thus, CHOOSENEXT in line 4 of Fig. 5 can always return a *next* node where FUZZ can continue.

If the recursive call does *not* return SUCCESSfully, it did not



(a) Attributed AST. (b) Previous FPs. (c) New FP.

Fig. 8: Construction of a generalized fail pattern (FP) F.

find a *next* sub-tree that satisfies all fail patterns, guards, and height limits, so the current sub-tree generated by the caller cannot be completed either. To make sure that the fuzzer does not make the same mistake again and to prune the remaining search space, NEWFAILPATTERN creates a new generalized fail pattern (see subsection IV-B4) that is added to the already learned fail patterns in *fps*. The next call of ALTERNATIVE then replaces the sub-tree with a new alternative (if possible).

4) Construction of New Generalized Fail Patterns: Fail patterns have a direct impact on the overall runtime. In principle, we could use full trees as fail patterns, but with such patterns that only exclude a single (already failed) alternative from further consideration, our algorithm would degenerate to naïve backtracking. The more wildcards a fail pattern has, the more alternatives it prunes from the search space. But overly general patterns may also exclude valid trees. Our algorithm would then discard sub-trees that could still be completed and it would hence waste the work spent to partially construct them. In the worst case, it would not find a valid tree at all.

We make use of the following observation to decide which nodes can be safely replaced by a wildcard, i.e., without excluding valid alternatives: When the recursive call for the *next* node FAILS[4] (lines 7–9 in Fig. 5), it could not find a valid tree *in the given context*. This context consists of both the fail patterns *next_fps* that the caller passed to the recursive call and the inherited attribute values of *next*. The key insight here is the following: If FUZZ is called again with the exact same context for *next*, the call would inevitably fail again; only if this context changes, the *next* construction *may* become possible. Thus, if changing a node N potentially also changes the context (by affecting the set *next_fps* or the inherited attribute values of *next*), we have to keep N in the fail pattern (a future call of ALTERNATIVE could then replace N to create a new context for *next*). If, however, changing N does *not* affect the context, N can be safely replaced by a wildcard.

Let us illustrate this with the example in Fig. 8 (in the AST, circles represent attributes and the dashed arrows depict their dependencies). Assume that the algorithm already learned the fail patterns F_1 and F_2 in Fig. 8(b) and passed the trimmed and filtered sub-patterns in the dashed boxes to the *next* call. When the call FAILS, we begin the construction of a new fail pattern by adding the non-wildcard nodes from these previous patterns (highlighted nodes in Fig. 8(b)):[5] The nodes a, b, d, and e must be included, since changing any of these nodes would also change the set *next_fps* (for example, if we changed e in the AST, F_1 would no longer match and its dashed sub-pattern would not be passed to the *next* call; this changes the context). We then add all nodes that the inherited attribute values of *next* directly or indirectly depend on (highlighted arrows in Fig. 8(a)), i.e., we also add the node f. The resulting fail pattern F is depicted in Fig. 8(c). Note that the node c legitimately becomes a wildcard: No matter how we change c, the *next* sub-tree could still not be completed.

As another example, reconsider Fig. 4. When the first call for the `UseVariable` sub-tree FAILS, the new fail pattern initially consists of all nodes on the chain to the root node. Since the inherited attributes of the failing node do not depend on the `Identifier` node, this nodes becomes a wildcard.

If the recursive call for *next* failed due to the height limit for recursive productions, the NEWFAILPATTERN only contains the chain from the *root* to the failing node (nodes a and d in the example). This forces ALTERNATIVE to replace the *next* node's parent d with an alternative and prevents our algorithm from running into the height limit over and over again.

TABLE I: LALA statistics and output after 72 hours.

	LALA lines	#generated programs	#nodes ($\cdot 10^9$)	total size (in GiB)	max. size (in KiB)	med. size (in KiB)
C	1 068	1 327 130	73.7	154.6	2 092	80.6
Lua	1 007	1 230 039	47.2	110.7	1 083	60.4
SQL-W	2 449	1 372 286	53.6	78.0	401	41.1
SQL-A	2 435	1 370 929	53.3	69.2	356	36.6
SMT	469	8 823 465	43.3	122.9	154	10.2
Csmith	—	546 042	—	56.1	1164	107.0

SQL-W: SQL with wrappers; SQL-A: SQL with arithmetic operations

V. EVALUATION

This section presents four case studies covering four different programming languages (*C*, *Lua*, *SQL*, and *SMT-LIB 2*). We evaluate ∗Smith both qualitatively and quantitatively: Our results show that ∗Smith can be used for a variety of quite different languages and testing scenarios and that it is able to efficiently generate huge test programs that uncover real bugs.

All (runtime) measurements were conducted on a single core of a workstation equipped with 128 GB of RAM and eight 2.4 GHz Intel Xeon CPUs running OpenJDK 8 on Debian 8.

A. Case Study 1: C

Our first case study targets *crashes* of *GCC* and *LLVM*, two open-source compilers for the *imperative*, *statically typed* C language. Our LALA specification (consisting of only 1 068 non-empty, non-comment lines; see Table I) ensures that the generated programs are compilable, but most of them cannot be executed without a runtime error.[6] We compare the

[4]The construction of fail patterns for a *root* node with failing guards (line 11 in Fig. 5) works analogously, but it suffices to consider the attribute dependencies of the guards; we omit details due to space constraints.

[5]If there were no previous fail patterns, we would initialize the fail pattern as the chain from the *root* node to the failing child (i.e., nodes a and d); this ensures that the new fail pattern can be matched against the *root* node.

[6]Our other case studies suggest that ruling out these runtime errors as well as all occurrences of undefined behavior is possible in principle, but due to the huge number of such cases in C [4] this is out of the scope of this paper.

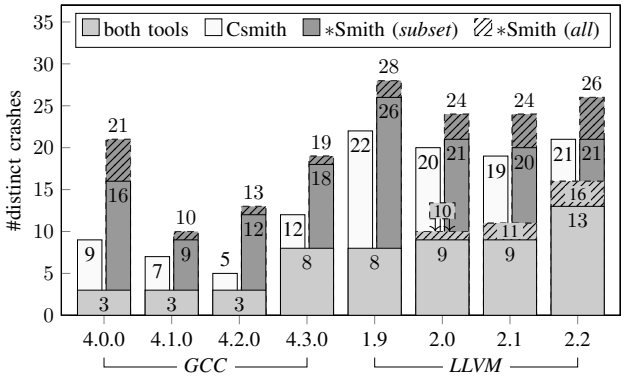


Fig. 9: Number of distinct *crashes* of *GCC* and *LLVM*.

efficiency and effectiveness of ∗Smith to that of Csmith [4].[7] Although Csmith can even guarantee that the generated programs are free of undefined behavior to enable RDT, it is also, to the best of our knowledge, the most efficient and effective tool for detecting C compiler crashes.

Like Csmith, we support arrays, pointers, `structs` and `unions`, `const` and `volatile` qualifiers, `gotos`, and labels. We also support constructs like recursive functions or overflows that Csmith disallows due to their undefined behavior.

To evaluate the *efficiency* of ∗Smith, we compare the output that it generates within 72 hours with that of Csmith. Table I holds the results: ∗Smith generates more programs ($2.43\times$) and achieves a higher total throughput ($2.75\times$) than Csmith. Moreover, ∗Smith generates even larger programs.

To show that ∗Smith is *effective*, we compiled the generated programs with different versions of two C compilers, *GCC* {4.0.0, 4.1.0, 4.2.0, 4.3.0} and *LLVM* {1.9, 2.0, 2.1, 2.2},[8] on five different optimization levels (`-O[0-3]`, `-Os`). We used a time limit of $10s$ per compilation as some runs appeared to be hanging in endless loops. In case of a compiler crash, both *GCC* and *LLVM* print an error message and location; similar to Yang et al. [4], we count different errors as different bugs.

Fig. 9 depicts the results, partitioned into bugs that were found by *both tools*, and bugs that were only found by Csmith or ∗Smith. To show that ∗Smith's higher effectiveness is not only due to its higher efficiency, we not only report the results for *all* 1 327 130 ∗Smith programs (hatched bars), but also for the *subset* consisting of the first 546 042 programs (solid bars). In total, the programs generated by Csmith trigger **115** different crashes (33 in *GCC* and 82 in *LLVM*). Even if we only consider the *subset*, the ∗Smith programs already trigger **143** different crashes (55 in *GCC* and 88 in *LLVM*). When we add the remaining programs, this number grows to **165** (63 in *GCC* and 102 in *LLVM*). These results not only suggest that ∗Smith is more effective in triggering compiler crashes than the language-specific Csmith, but also emphasize the importance of a highly efficient fuzzer.

[7]We used the most recent version 2.3 and configured Csmith to omit comments (for a fairer comparison of program sizes) and to not emit code that computes the hash sums of global variables (only needed for *RDT*).

[8]We used older versions of both compilers due to their higher bug density compared to newer versions [4]. This allows for a fairer comparison.

Note that only about 28% of all crashes were triggered by both tools. We attribute this to the differences in the generated programs: While Csmith mainly generates nested calls of helper functions, ∗Smith generates programs that make heavy use of arithmetic expressions. This enables different optimization techniques, which in turn leads to different bugs.

B. Case Study 2: Lua

In the remaining three case studies, we not only consider compiler *crashes*, but also search for *wrong results* via RDT: We feed the generated programs into all implementations, compare the outputs, and count deviations from the majority vote[9] as a wrong result. This requires that the generated programs are free of undefined behavior.

Our LaLa specification for the *dynamically typed*, *imperative* language Lua supports most features of Lua 5.1 [40] (this is the version supported by the *LuaJIT* implementation [41] that we wanted to include). As far as we know, there is no generally applicable approach yet to create type-correct programs in a dynamically typed language (even the recent *CodeAlchemist* [42] for JavaScript cannot guarantee this). We could treat Lua like a statically typed language, but this would severely limit the expressiveness of the generated programs. We thus propose a more sophisticated approach: A variable (or table member) may hold values of different types on different control flow paths; when these paths meet, we merge the types to a common type. (Although our LaLa rules allow nested control structures and even `break` statements that may bypass variable definitions, we found that these computations can be mapped to attribute rules quite easily.) Whenever a variable is redefined, we ensure that the new type is *compatible* to the old one (e.g., in Lua a `number` is compatible to a `boolean` value, because the former can be used in all places where the latter can be used). This way, we can ensure that all variable usages are type-correct and still allow for variables whose type changes dynamically.

Like C, Lua has *undefined behavior*. For example, a Lua implementation may freely choose the evaluation order of expression lists. In case of side effects, this may lead to differing program results (which prevents RDT). Moreover, the range of Lua's `numbers` is *implementation-defined*. Thus, to enable RDT, our LaLa rules exclude all undefined behavior and we made sure that all compared implementations use the same `number` range. To easily compare the executions of different implementations, the generated programs print pseudo-random numbers when entering blocks, as well as the values of all global variables at the end of the program.

Despite the higher requirements that our LaLa rules impose on the Lua programs to rule out all undefined behavior, the throughput of ∗Smith is almost as high as for C, see Table I. This also supports our claim that ∗Smith is truly language-agnostic and generalizes to languages that differ clearly.

We used the generated programs to search for bugs in different Lua implementations. Besides all versions of the reference

[9]As always with RDT, the majority could be wrong. To mitigate this risk, we manually analyzed some of the failure-inducing test programs, see Sec. V-E.

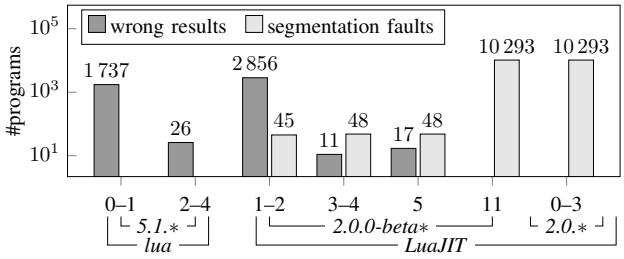


Fig. 10: Number of *wrong results* and *segmentation faults* in different Lua implementations (logarithmic plot).

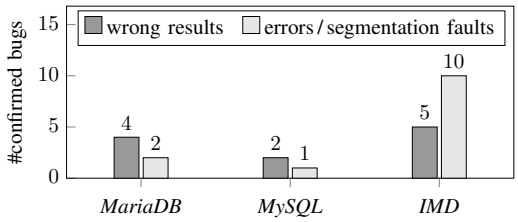


Fig. 11: Unique bugs per SQL database that ∗Smith uncovered and that have been confirmed; no bugs found in *PostgreSQL*.

implementation, *lua 5.1.[0-5]*, we also considered all publicly available versions of the alternative *LuaJIT* implementation, *LuaJIT 2.0.0-beta[1-11]*, *2.0.[0-5]*, and *2.1.0-beta[1-3]*.

In preliminary experiments we found that some of the generated programs contain an endless loop or produce huge outputs. We thus introduced a time limit of $5s$ per execution and limited the size of the outputs to 1 MiB (higher thresholds could only increase the number of uncovered bugs). We also discovered that the different versions disagree whether or not to distinguish between $+0$ and -0 when printing variable values. Since the correct behavior is questionable,[10] we filtered these values when comparing the different outputs.

Fig. 10 depicts the results. The Lua programs generated within 72 hours trigger *wrong results* in **5** different versions of *lua*, as well as *wrong results* in **5** and *segmentation faults* in **10** different versions of *LuaJIT*. While we did not find any bugs in the most recent versions, the results still show the effectiveness of ∗Smith: Older versions of *lua* and *LuaJIT* contained (one or more) bugs that could have easily been detected with ∗Smith.

C. Case Study 3: SQL

Third, we cover the *statically typed*, *declarative* database query language SQL. The generated queries are self-contained, i.e., they contain `schema`, `table`, and `view` definitions, as well as `insert` operations. The queries may also contain (nested) `select` statements, which may involve compound filtering conditions, `joins`, `group by` operations, as well as aggregate and window functions (e.g., `count` or `max`).

Unfortunately, since there is *implementation-defined* behavior in SQL (for example w.r.t. the internal representation of strings or the handling of `NULL` values), database implementations differ considerably. As this hampers RDT, our LaLa rules exclude all language features that the tested implementations handle differently. One notable exception is the size of numeric data types: The databases under test use different sizes, but excluding all numeric values would considerably limit the expressiveness of the generated queries. Our first approach (*SQL-W*) to still compare different implementations is inspired by Csmith [4] and replaces all arithmetic operations with *w*rapper functions. These functions check if an operation causes an overflow in any of the tested databases, in which case they return a dummy value. Unfortunately, this prevents many query optimizations and hides optimization-related bugs. Our second approach (*SQL-A*) thus allows *a*rithmetic operations that possibly overflow. While this leads to some erroneous queries, the databases report these overflows at runtime; we can thus easily detect and discard such queries.

Table I shows the throughput for both variants. In comparison with the other languages, ∗Smith generates a similar number of programs and AST nodes. But the the total size of the generated files is slightly lower since the complexity of SQL (and hence our LaLa specifications) leads to complex, deep ASTs: While the median height of the generated SQL ASTs is 55, the median height of the C ASTs (the second highest value in our case studies) is only 32.

We searched for bugs in four different databases: *PostgreSQL* [43], *MariaDB* [44], *MySQL* [45], and a commercially available *i*n-*m*emory *d*atabase (called *IMD* below). When we started fuzzing these databases, ∗Smith quickly uncovered bugs. While this was delightful (at least for us), this also complicated the search for more bugs. Some of the initial bugs were triggered so often, that manually inspecting the failure-inducing queries to check for other bugs quickly became infeasible. We thus reported the bugs and temporarily excluded the failure-inducing constructs from our LaLa specifications (i.e., we used subsets of *SQL-W* and *SQL-A* to search for further bugs). Therefore, we do not evaluate the effectiveness of ∗Smith with the queries from Table I, but Fig. 11 presents the number of confirmed bug reports instead.

Only in *PostgreSQL* ∗Smith did not find any bugs. For *MariaDB*, we reported **6** unique bugs (4 *wrong results*, 2 invalid runtime *errors* or *segmentation faults*) that have already been confirmed (**1** more *wrong result* is currently pending). **3** of these bugs (plus the **1** pending bug) also occur in *MySQL* (which *MariaDB* was initially forked from). In *IMD*, ∗Smith uncovered **15** confirmed bugs (5 *wrong results*, 10 *errors* or *segmentation faults*). This higher number is a consequence of the *IMD* developers responding quickly and providing patched versions during our evaluation. This allowed us to detect additional, previously hidden bugs.

D. Case Study 4: SMT-LIB 2

SMT-LIB 2 [46] is the *statically typed*, *declarative* language that most competitive SMT solvers support to specify first-order logic formulas. Since these are often used for program verification tasks, their correctness is important [47], [48].

SMT solvers typically support different *logics* that define which data types and language constructs may be used. We picked the `QF_UFBV` logic that supports *uninterpreted functions*

[10] `http://lua-users.org/lists/lua-l/2011-06/msg00188.html`.

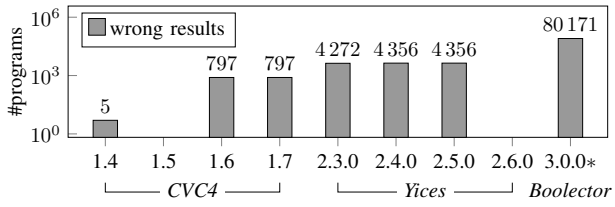


Fig. 12: *Wrong results* (*sat* vs. *unsat*) in the different SMT solver versions (logarithmic plot); no bugs found in *z3*. We only found *segmentation faults* in *CVC4 1.4* (triggered by more than half of our SMT scripts) and thus omit these bugs here.

and *bit vectors* (of arbitrary size) without quantifiers, since this logic is supported by multiple different solvers and suffices for many program verification tasks. The LaLa specification supports most of the `QF_UFBV` features, including functions, assertions, `push`/`pop` commands, `let` expressions, `extract` and `concat` operations, as well as the boolean and bit vector expressions defined by the language standard. To enable RDT, we avoid the undefined behavior of the division and remainder operations for divisors of 0 by guarding these with an `ite` (*if-then-else*) operation. Other than this, there is no undefined behavior: Formulas are either satisifiable (*sat*) or not (*unsat*).

As SMT scripts describe logical conjunctions, the probability that a random formula is *unsat* increases with its size. This decreases the chances to find bugs. We thus chose production weights that lead to smaller (but more) programs, see Table I. While the throughput is still high, it is slightly lower than for the other languages, since SMT-LIB 2 is *strongly typed* and thus imposes many restrictions on the generated scripts.

We used the generated SMT scripts to search for bugs in all four solvers that participated in the `QF_UFBV` track of the most recent SMT competition, *SMT-COMP 2019* [49]. For *z3* [50], *CVC4* [51], and *Yices* [52] we picked the respective current version as well as several older versions. For *Boolector* [53] the situation is slightly more intricate. The latest release, *Boolector 3.0.0*, erroneously rejects many scripts due to **2** bugs that ∗Smith uncovered in preliminary experiments. We reported these bugs and both of them have been fixed by now. Below, *Boolector 3.0.0∗* refers to a patched version. Due to the bugs, we had to exclude older versions of *Boolector*.

Fig. 12 shows the number of *wrong results* that the generated scripts uncovered in the solvers via RDT.While we did not observe any cases where *z3* computed the wrong result, we found bugs in at least one version of all other solvers.

For *Yices*, we found bugs in older versions that apparently have been fixed by now. 4 267 of the 4 272 scripts for which *Yices 2.3.0* yields the wrong result also trigger a bug in *Yices 2.4.0* and *2.5.0*. This indicates that one or more bugs stayed unnoticed across several releases (spanning more than 17 months) that could have easily been detected with ∗Smith. For *CVC4* and *Boolector*, the situation is even more severe, as we found bugs in the current versions of both solvers.

Upon further investigation, we found that the wrong results of *CVC4* and *Boolector* are likely due to **1** bug in each solver. We reported both bugs to the respective developers. In both cases, the developers confirmed and fixed the bugs. Thus, in total, we reported **4** bugs that have been confirmed and fixed.

This shows that ∗Smith is a valuable tool that can detect severe, previously unknown bugs, even in supposedly well tested (since safety-critical) applications like SMT solvers.

E. Threats to Validity

Since we are by no means experts for any of the four programming languages that we used to evaluate ∗Smith, it is possible that our LaLa specifications do not fully conform to the respective language definition. While *crashes* and *segmentation faults* are clear bugs, the case is more intricate for bugs that manifest as *wrong results*. In principle, it is possible that we erroneously blamed a compiler due to an invalid input program (or, similarly, that a majority vote was wrong and that we falsely punished the minority). To mitigate this risk, for each compiler version that our evaluation flagged defective we manually reduced and analyzed several failure-inducing test programs. We did not observe a single case in which the respective compiler was actually correct. Also note that we reported all bugs that we found in any of the current compiler versions; all of these bugs (except for a pending one) have been confirmed by the respective developers.

We had to manually patch the older *GCC* and *LLVM* versions to build them with a modern toolchain. This may have introduced bugs. But since our Csmith results closely resemble those of its original evaluation [4], we are confident that the impacts of our patches are negligible.

Our results could differ both quantitatively and qualitatively, had we used other random seeds, production weights, or height limits. However, most of the bugs that we found were triggered by more than one program. Chances are high that these bugs would have also been found with different configurations.

VI. Conclusion

We presented ∗Smith, a novel framework for the generation of *compilable* test programs in *arbitrary*, *real* programming languages. Users of ∗Smith do not need to implement the program generation logic. Instead, they provide a specification in a newly designed language LaLa that captures the syntactic and semantic rules of the respective programming language in a concise, declarative way that is inspired by attribute grammars. Key to our approach is a new fuzzing algorithm that uses novel technical ideas (only *local* modifications in the AST; *fail patterns* to prune the search space) to achieve a high throughput and to generate large, complex test programs.

Four case studies showed that ∗Smith is *flexible*, *efficient*, and *effective*, and hence a valuable tool for the developers of compilers and other language processors.

∗Smith and our example specifications are open-source (`https://github.com/FAU-Inf2/StarSmith`).

References

[1] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An Empirical Comparison of Compiler Testing Techniques," in *ICSE'16: International Conference on Software Engineering*, Austin, TX, May 2016, pp. 180–190.

[2] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[3] V. Le, M. Afshari, and Z. Su, "Compiler Validation via Equivalence Modulo Inputs," in *PLDI'14: Programming Language Design and Implementation*, Edinburgh, United Kingdom, June 2014, pp. 216–226.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *PLDI'11: Programming Language Design and Implementation*, San Jose, CA, June 2011, pp. 283–294.

[5] P. Purdom, "A Sentence Generator for Testing Parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, 1972.

[6] MozillaSecurity, "dharma," Accessed on 2019-10-09, https://github.com/MozillaSecurity/dharma.

[7] M. Zimmermann, "Tavor," Accessed on 2019-10-09, https://github.com/zimmski/tavor.

[8] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[9] C. Lindig, "Random Testing of C Calling Conventions," in *AADEBUG'05: International Symposium on Automated And Analysis-Driven Debugging*, Monterey, CA, Sep. 2005, pp. 3–12.

[10] E. Eide and J. Regehr, "Volatiles Are Miscompiled, and What to Do about It," in *EMSOFT'08: International Conference on Embedded Software*, Atlanta, GA, Oct. 2008, pp. 255–264.

[11] E. Nagai, A. Hashimoto, and N. Ishiura, "Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers," in *SASIMI'13: Workshop on Synthesis And System Integration of Mixed Information Technologies*, Sapporo, Hokkaido, Japan, Oct. 2013, pp. 88–93.

[12] F. Sheridan, "Practical testing of a C99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007.

[13] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-Core Compiler Fuzzing," in *PLDI'15: Programming Language Design and Implementation*, Portland, OR, June 2015, pp. 65–76.

[14] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated Testing of Graphics Shader Compilers," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 93:1–93:29, Oct. 2017.

[15] MozillaSecurity, "jsfunfuzz," Accessed on 2018-10-30, https://github.com/MozillaSecurity/funfuzz/.

[16] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random Program Generator for Java JIT Compiler Test System," in *QSIC'03: International Conference on Quality Software*, Dallas, TX, Nov. 2003, pp. 20–24.

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *USENIX Security Symposium*, Bellevue, WA, Aug. 2012, pp. 445–458.

[18] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming," in *ESORICS'16: Computer Security*, ser. Springer LNCS vol. 9878, Heraklion, Greece, Sep. 2016, pp. 581–601.

[19] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler Fuzzing through Deep Learning," in *ISSTA'18: International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, July 2018, pp. 95–105.

[20] E. G. Sirer and B. N. Bershad, "Using Production Grammars in Software Testing," in *DSL'99: Domain-Specific Languages*, Austin, TX, Oct. 1999, pp. 1–13.

[21] Q. Zhang, C. Sun, and Z. Su, "Skeletal Program Enumeration for Rigorous Compiler Testing," in *PLDI'17: Programming Language Design and Implementation*, Barcelona, Spain, June 2017, pp. 347–361.

[22] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an Optimising Compiler by Generating Random Lambda Terms," in *AST'11: International Workshop on Automation of Software Test*, Waikiki, HI, May 2011, pp. 91–97.

[23] G. Barany, "Finding Missed Compiler Optimizations by Differential Testing," in *CC'18: Compiler Construction*, Vienna, Austria, Feb. 2018, pp. 82–92.

[24] C. Sun, V. Le, and Z. Su, "Finding and Analyzing Compiler Warning Defects," in *ICSE'16: International Conference on Software Engineering*, Austin, TX, May 2016, pp. 203–213.

[25] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing Static Analyzers with Randomly Generated Programs," in *NFM'12: International Conference on NASA Formal Methods*, Norfolk, VA, Apr. 2012, pp. 120–125.

[26] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *ASE'17: Automated Software Engineering*, Urbana-Champaign, IL, Oct.–Nov. 2017, pp. 590–600.

[27] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *ESEC/FSE'07: European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep. 2007, pp. 185–194.

[28] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[29] G. Misherghi and Z. Su, "HDD: Hierarchical Delta Debugging," in *ICSE'06: International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 142–151.

[30] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-Case Reduction for C Compiler Bugs," in *PLDI'12: Programming Language Design and Implementation*, Beijing, China, June 2012, pp. 335–346.

[31] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-Guided Program Reduction," in *ICSE'18: International Conference on Software Engineering*, Gothenburg, Sweden, May–June 2018, pp. 361–371.

[32] S. Herfert, J. Patra, and M. Pradel, "Automatically Reducing Tree-Structured Test Inputs," in *ASE'17: International Conference on Automated Software Engineering*, Urbana-Champaign, IL, Oct.–Nov. 2017, pp. 861–871.

[33] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming Compiler Fuzzers," in *PLDI'13: Programming Language Design and Implementation*, Seattle, WA, June 2013, pp. 197–208.

[34] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test Case Prioritization for Compilers: A Text-Vector Based Approach," in *ICST'16: Software Testing, Verification and Validation*, Chicago, IL, Apr. 2016, pp. 266–277.

[35] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to Prioritize Test Programs for Compiler Testing," in *ICSE'17: International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017, pp. 700–711.

[36] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[37] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A Verified Implementation of ML," in *POPL'14: Principles of Programming Languages*, San Diego, CA, Jan. 2014, pp. 179–191.

[38] D. E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968, correction: *Mathematical Systems Theory*, vol. 5, no. 2, pp. 95-96, 1971.

[39] D. Grune, K. v. Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern Compiler Design*, 2nd ed. New York: Springer Publishing, 2012.

[40] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "Lua 5.1 Reference Manual," Accessed on 2019-10-03, https://www.lua.org/manual/5.1/.

[41] M. Pall, "The LuaJIT Project," Accessed on 2019-10-03, https://luajit.org/.

[42] H. Han, D. Oh, and S. K. Cha, "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines," in *NDSS'19: Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2019.

[43] "PostgreSQL: The world's most advanced open source database," Accessed on 2019-10-06, https://www.postgresql.org/.

[44] "About MariaDB - MariaDB.org," Accessed on 2019-10-06, https://mariadb.org/about/.

[45] "MySQL," Accessed on 2019-10-06, https://www.mysql.com/de/.

[46] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard Version 2.6," Department of Computer Science, The University of Iowa, Iowa City, IA, Tech. Rep., 2017.

[47] A. Niemetz and A. Biere, "ddSMT: A Delta Debugger for the SMT-LIB v2 Format," in *SMT'13: International Workshop on Satisfiability Modulo Theories*, Helsinki, Finland, July 2013, pp. 36–45.

[48] R. Brummayer and A. Biere, "Fuzzing and Delta-Debugging SMT Solvers," in *SMT'09: International Workshop on Satisfiability Modulo Theories*, Montreal, Canada, Aug. 2009, pp. 1–5.

[49] "SMT-COMP 2019: QF_UFBV (Incremental Track)," Accessed on 2019-10-03, https://smt-comp.github.io/2019/results/qf-ufbv-incremental.
[50] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems*, ser. Springer LNCS vol. 4963, Budapest, Hungary, Mar.–Apr. 2008, pp. 337–340.
[51] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV'11: Computer Aided Verification*, ser. Springer LNCS vol. 6806, Snowbird, UT, July 2011, pp. 171–177.
[52] B. Dutertre, "Yices 2.2," in *CAV'14: Computer Aided Verification*, ser. Springer LNCS vol. 8559, Vienna, Austria, July 2014, pp. 737–744.
[53] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 System Description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2015.