Input Invariants

Dominic Steinhöfel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
dominic.steinhoefel@cispa.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

ABSTRACT

How can we generate valid system inputs? Grammar-based fuzzers are highly efficient in producing *syntactically* valid system inputs. However, programs will often reject inputs that are *semantically* invalid. We introduce ISLa, a *declarative specification language for context-sensitive properties* of structured system inputs based on context-free grammars. With ISLa, it is possible to specify *input constraints* like "a variable has to be defined before it is used," "the 'file name' block must be 100 bytes long," or "the number of columns in all CSV rows must be identical."

Such constraints go into the ISLa *fuzzer,* which leverages the power of solvers like Z3 to solve semantic constraints and, on top, handles quantifiers and predicates over grammar structure. We show that a few ISLa constraints suffice to produce 100% semantically valid inputs while still maintaining input diversity. ISLa can also *parse* and precisely *validate* inputs against semantic constraints.

ISLa constraints can be *mined* from existing input samples. For this, our ISLearn prototype uses a catalog of common patterns, instantiates these over input elements, and retains those candidates that hold for the inputs observed and whose instantiations are fully accepted by input-processing programs. The resulting constraints can then again be used for fuzzing and parsing.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Specification languages*; **Constraint and logic languages**; *Syntax*; *Semantics*; **Parsers**; *Software reverse engineering*; *Documentation*; • **Theory of computation → Grammars and context-free languages**; *Formalisms*.

KEYWORDS

fuzzing, specification language, grammars, constraint mining

1 INTRODUCTION

Automated software testing with random inputs (*fuzzing*) [20] effectively finds bugs in programs. Pure random inputs can quickly discover errors in input processing. Yet, if a program expects complex structured inputs (e.g., C programs, JSON expressions, or binary formats), the chances of randomly producing *valid inputs* that are accepted by the parser and reach deeper functionality are low.

Language-based fuzzers [9, 13, 14] overcome this limitation by generating inputs from a *specification* of a program's expected input language, frequently expressed as a *Context-Free Grammar (CFG)*. This considerably increases the chance of producing an input passing the program's parsing stage and reaching its core logic. Yet, while being great for *parsing, CFGs are often too coarse for* producing *inputs.* Consider, e.g., the language of XML documents (without document type). In XML, the identifiers of opening and closing tags have to match each other, as in "\<name\>text\</name\>." Such matching cannot be expressed in a CFG. Even though a CFG can be used for parsing (cf. Fig. 1), using it as a *producer* for fuzzing will typically yield non-matching inputs like "\<O L="cmV"\>B7\</P\>." Such invalid inputs are still useful for *testing the parser,* but hardly ever reach functionality beyond input rejection.

To allow for precise production, we can switch to a *different formalism.* However, existing solutions all have their drawbacks. Using *general purpose code* to produce inputs or enriching grammars with such code is closely tied to an implementation language, and does not allow for *parsing* and *recombining* inputs, which is a common feature of modern fuzzers. *Unrestricted grammars* can in principle specify any computable input property, but we see them as "Turing tar-pits," in which "everything is possible, but nothing of interest is easy" [23]—just try, for instance, to express that some number is the sum of two input elements.

In this paper, we bring forward a different solution by introducing a (programming and target) *language-independent,* declarative specification language named *ISLa* (**I**nput **S**pecification **La**nguage) for expressing *semantic constraints over CFGs.* By enriching existing grammars with constraints, we leverage the simplicity of CFGs, while significantly extending their expressiveness. ISLa is designed as a *general-purpose formalism* to specify the format even of highly complex program inputs. It can be used to *produce* such inputs automatically, making ISLa a *highly effective fuzzer that can pass all validation steps*; and it can be used to *parse* inputs and *validate* constraints, making ISLa an effective *validator* of inputs.

To formalize an input format in and for ISLa, one starts with the definition of a CFG. If a grammar is not already available, it is possible to extract it from inputs [18] and programs [10]. Then, one iteratively *strengthens the definition* by adding more and more ISLa constraints until the represented language is a sufficiently close approximation of the target language—an *invariant* over all inputs.

$\langle xml\text{-}tree \rangle ::= \langle xml\text{-}openclose\text{-}tag \rangle$
$\qquad | \; \langle xml\text{-}open\text{-}tag \rangle \; \langle inner\text{-}xml\text{-}tree \rangle \; \langle xml\text{-}close\text{-}tag \rangle$
$\langle inner\text{-}xml\text{-}tree \rangle ::= \langle TEXT \rangle \; | \; \langle xml\text{-}tree \rangle$
$\qquad | \; \langle inner\text{-}xml\text{-}tree \rangle \; \langle inner\text{-}xml\text{-}tree \rangle$
$\langle xml\text{-}open\text{-}tag \rangle ::= \; `<`\; \langle id \rangle \; `>` \; | \; `<`\; \langle id \rangle \; `\_`\; \langle xml\text{-}attribute \rangle \; `>`$
$\langle xml\text{-}close\text{-}tag \rangle ::= \; `</`\; \langle id \rangle \; `>`$
$\langle xml\text{-}openclose\text{-}tag \rangle ::= \; `<`\; \langle id \rangle \; `/>` \; | \; `<`\; \langle id \rangle \; `\_`\; \langle xml\text{-}attribute \rangle \; `/>`$
$\langle xml\text{-}attribute \rangle ::= \; \langle id \rangle \; `="`\; \langle TEXT \rangle \; `"`$
$\qquad | \; \langle xml\text{-}attribute \rangle \; `\_`\; \langle xml\text{-}attribute \rangle$
$\langle id \rangle ::= \langle id\text{-}no\text{-}prefix \rangle \; | \; \langle id\text{-}with\text{-}prefix \rangle$
$\langle id\text{-}no\text{-}prefix \rangle ::= \langle ID\text{-}START\text{-}CHAR \rangle \; \langle ID\text{-}CHAR \rangle *$
$\langle id\text{-}with\text{-}prefix \rangle ::= \langle id\text{-}no\text{-}prefix \rangle \; `:`\; \langle id\text{-}no\text{-}prefix \rangle$

Figure 1: A context-free grammar for XML. Lexer rules (for the capitalized nonterminals) are omitted.

```
<xml-tree>.<xml-open-tag>.<id> = <xml-tree>.<xml-close-tag>.<id>
```

Listing 1: ISLa constraint for well-balanced XML.

```
forall <xml-tree> tree="<{<id> opid}[<xml-attribute>]><inner-xml-
    ↪ tree></{<id> clid}>" in start:
  (= opid clid)
```

Listing 2: Core-ISLa constraint for well-balanced XML.

To get an idea of ISLa constraints, consider Listing 1, referring to the grammar in Fig. 1. The constraint expresses that the $\langle id \rangle$s of opening and closing tags—in *all* XML trees that are not self-closing—are equal. Programmers write such constraints in the *simplified* layer of the ISLa language. Our solver translates it to the "Core-ISLa" constraint in Listing 2. The Core-ISLa version *explicitly quantifies* over all $\langle xml\text{-}tree \rangle$ elements, binds its constituents to variables using *pattern matching* and expresses the equality using the *SMT-LIB formula* [1] "(= opid clid)." This is typical for an ISLa constraint: It first *identifies* derivation tree elements, and then poses *constraints* over these elements. During fuzzing, ISLa then would produce matching pairs of opening and closing $\langle id \rangle$s.

The resulting valid inputs can be used as *seed inputs* for mutational fuzzers like AFL. The ISLa solver can be integrated into the *fitness function* of evolutionary fuzzers, guiding their mutations toward semantically valid inputs; the solver can quickly reject invalid inputs without having to run actual tests.

Yet, where would ISLa *constraints* come from? Testers can write ISLa constraints manually, thus ensuring input validity, and add additional constraints to further control the inputs generated. However, they can also *mine constraints from existing inputs.* To this end, our *ISLearn* tool uses a catalog of common *constraint patterns,* instantiates these over all inputs and input elements, and retains those constraint candidates that hold for all inputs. The catalog holds patterns to identify matching elements, length relations, arithmetic relations, checksums, and more. ISLearn is similar in spirit to the *Daikon* function-level invariant detector [7]. On top, ISLearn can *verify* and *refine* constraint candidates by having the program under test check whether derived concrete inputs are valid.

After illustrating ISLa by example (Section 2), this paper makes the following contributions:

A specification language for input constraints. We propose a formalism (ISLa) for augmenting existing context-free grammars with context-sensitive constraints. ISLa has a rich *declarative* layer, separating semantic properties (constraints) from syntactic properties (the grammar). We formally define its syntax and semantics in Section 3. To the best of our knowledge, ISLa is the *first formalism to express context-sensitive constraints for the generation of system inputs.*

Semantic fuzzing. We describe an efficient procedure to generate inputs satisfying ISLa constraints (and their grammars), and discuss our implementation (Section 4). To the best of our knowledge, ISLa is the *first fuzzer (and checker) to make use of such constraints,* giving users unprecedented means to specify which system inputs should be generated.

Mining input constraints. We introduce ISLearn, a system for automatically mining input constraints in conjunctive normal form based on a configurable pattern catalog (Section 5). To the best of our knowledge, ISLearn is the first approach to *infer such invariants from given system inputs.*

ISLa and its constraints are effective. In our evaluation (Section 6), we formalize semantic properties from diverse languages, namely XML, a subset of C, reStructuredText, CSV files, and TAR archives. Our results demonstrate that already a few lines of ISLa specifications suffice to generate *100% precise* inputs while maintaining *diversity.* On top, our constraint miner ISLearn can extract precise invariants about ICMP packets, DOT graphs, and Racket programs. After discussing related work (Section 7), Section 8 closes with conclusion and future work. Our electronic appendix [25] provides further formalizations, explanations, examples, and proofs.

2 ISLA BY EXAMPLE

Let us illustrate the expressive power of ISLa by detailing our XML example. When randomly feeding an XML processor (e.g., Python's `xml.etree` package) with inputs generated from the XML grammar in Fig. 1 using a grammar fuzzer, we obtain not only one, but *three* kinds of errors: (1) "Mismatched tag," (2) "duplicate attribute," and (3) "unbound prefix." By adding ISLa specifications to the XML grammar, we can substantially increase the portion of valid XML we pass to the processor. Moreover, these specifications document XML features relevant to the parser of our test target.

Since ISLa is closed under conjunction, we can *incrementally* refine the specification simply by adding individual input constraints until we are satisfied with the quality of the generated inputs or the value of the specification as a documentation measure.

From the invalid inputs generated from the XML grammar, about 52% are invalid due to a mismatched tag, and about 22% because of an unbound prefix. Let us address these.

2.1 Matching Tags

The ISLa constraints in Listings 1 and 2 addresses the problem of *mismatched tags* by enforcing that the two IDs match. The Core-ISLa version (Listing 2) uses a *universal quantifier* (**forall**) over all sub expressions of type $\langle xml\text{-}tree \rangle$, which is the specified type of the bound variable tree. Types are nonterminals from the reference grammar (here the XML grammar in Fig. 1) or the special type int for quantifiers over numbers. The present quantifier uses

pattern matching. ISLa only considers matches conforming to the pattern (in quotation marks); in the case of a successful match, not only the quantified variable tree but also the variables opid and clid in the pattern (in curly braces) are *bound* to the corresponding parts of the matched input segment. Match expressions may contain optional elements in square brackets to capture multiple expansion alternatives. The core of the **forall** formula is an *SMT-LIB S-expression* stating that opid and clid are equal. Since ISLa extends the SMT-LIB language [29], it supports all its string constraints. A Core-ISLa constraint contains exactly one constant symbol, which determines the type of described inputs. By default, this is a symbol start of type ⟨*start*⟩, which can be customized by a declaration "**const** name: type;" before the actual constraint.

Simplified ISLa. ISLa's simplified language layer allows us to write this constraint much more concisely. Listing 1 is the simplified version of the Core-ISLa constraint in Listing 2. In simplified ISLa, we can use the more common, mathematical infix syntax for binary expressions, i.e., "x = y" instead of "(= x y)." Furthermore, the **forall** quantifier can be omitted. Instead, we directly use its type ⟨*xml-tree*⟩ in the equation. This implicitly adds a **forall** quantifier over ⟨*xml-tree*⟩ elements. To access *immediate* children of ⟨*xml-tree*⟩ elements, we use a notation inspired by the XPath abbreviated syntax [3]. For example, <xml-tree>.<xml-open-tag> refers to the *first* ⟨*xml-open-tag*⟩ element among the children of an ⟨*xml-tree*⟩ element in a derivation tree, if any.

In simplified ISLa, "**in** start" in quantifiers is a default and can be omitted. Furthermore, variable names in quantifiers, such as var in "**forall** <type> var="...": ...," can be omitted if we are only interested in the variables bound by the match pattern. Simplified and Core-ISLa syntax can be *mixed* in the same constraint. In the end, all constraints are *translated to Core-ISLa*.

Since ISLa constraints are closed under conjunction (**and**) and disjunction (**or**), it is easy to refine (or relax) constraints. ISLa is thus well suited for *targeted* testing, or, e.g., for describing a *specific class of inputs* that trigger a bug in a debugging scenario. Thanks to its declarative nature, it can also be used for formulating human-readable *specifications* of the expected inputs of a system.

2.2 Binding Prefixes

Next, we specify a property avoiding "unbound prefix" errors. An "unbound prefix" error is raised when tag or attribute identifiers in XML documents contain a *namespace prefix*, such as ns1 and ns2 in "<ns1:tag ns2:attr="..."/>," which is not *declared* in the same or an outer tag. This is an example of a *def-use* property that is also common in programming languages: A *used* identifier must be *defined* in some outer scope or at some preceding position. One adds the attribute "xmlns:ns1="some_text"" to declare namespace ns1, where frequently, the quoted text contains a URL. The property we aim for is expressed more precisely as: "*For all* identifiers with a prefix *p*, *there is* a *surrounding* XML tree *t* such that *there is* an attribute xmlns:*p* *in* the attributes list of *t*'s opening tag." We emphasized words corresponding to ISLa language elements. There is one subtlety, though: We have to distinguish prefixes in attribute and tag identifiers, since the special attribute xmlns does *not* have to be declared, as it is used precisely to declare other namespaces.

Again, we can express both cases in isolation to incrementally refine the specification. Here, we regard the slightly more complicated case of prefixes in attribute identifiers. Listing 3 shows the ISLa specification for this case.

```
1 forall <xml-attribute> attr="{<id-no-prefix> prefix_use}:{<id-no-
    ↪ prefix> maybe_def}=\"<text>\"": (
2   not prefix_use = "xmlns" implies
3   exists <xml-tree> outer_tag=
4     "<<id> {<xml-attribute> a_cont}><inner-xml-tree></<id>>": (
5   inside(attr, outer_tag) and
6   exists <xml-attribute>=
7     "xmlns:{<id-no-prefix> prefix_def}=\"<text>\"" in a_cont:
8   (not prefix_def = "xmlns" and prefix_use = prefix_def)))
```

Listing 3: ISLa constraint for binding prefixes in attribute identifiers (reference grammar: Fig. 1)

The ISLa code closely resembles the natural language specification we described previously, except that we specialized it to only quantify over *attributes* (Line 1) and generally permit the xmlns prefix (Line 2) using an *implication*: Only *if* the prefix is *not* xmlns, it must be explicitly defined.

2.3 Targeted Testing

With ISLa specifications, we can go beyond constraints for semantic validity for *application-specific, targeted testing*. Imagine an XML processor that allows associating tags with URLs defined using dedicated attributes web:baseurl and web:query for base URLs and query strings. We can enforce the existence of a tag using both of these attributes somewhere in any produced system input:

```
exists <xml_attribute> attributes: (
  exists <xml-attribute> attr in attributes:
    attr.<id> = "web:baseurl" and
  exists <xml-attribute> attr in attributes:
    attr.<id> = "web:query")
```

The XML processor performs some input validation and rejects all inputs where the values of these attributes exceed a length of 100 characters. We force all generated inputs to respect this constraint by adding the following specification:

```
forall <xml-attribute>="web:<id-no-prefix>={<text> text}":
  str.len(text) <= 100
```

After parsing an XML file, the processor assembles a complete URL by joining the base URL and the query string. However, let us assume its input validation is buggy: The result is stored in a character array of length 150, and we thus get a *buffer overflow* when the base URL and the query string *together* exceed a length of 150 characters. We can then *explicitly generate* inputs triggering this bug by encoding this property as an ISLa constraint. Such inputs would be valuable for developers or security researchers, as a regression test validating a fix for a potential exploit:

```
forall <xml-attribute> attrs:
  forall <xml-attribute> attr_1="web:baseurl={<text> t1}" in attrs:
    forall <xml-attribute> attr_2="web:query={<text> t2}" in attrs:
      str.len(t1) + str.len(t2) > 150
```

2.4 Mining Constraints

Constraints like the ones described above can also be *mined* from existing inputs. To mine constraints such as the XML constraint in

Listing 2, we create a *schematic* version of it that is *independent of the choice of a particular grammar:*

```
forall <?NONTERMINAL>="{<?MATCHEXPR(opid, clid)>}": opid = clid
```

This pattern can be added to the catalog of our ISLearn system, enabling the system to infer similar constraints for a different grammar. The placeholder `<?NONTERMINAL>` represents any nonterminal in that grammar; `<?MATCHEXPR(opid, clid)>` represents any suitable match expression for an instantiation of `<?NONTERMINAL>`, containing two nonterminal occurrences that are bound to variables `opid` and `clid`. ISLearn generates candidate instantiations from such patterns and then filters those that hold for a set of given or automatically generated sample inputs. Hence, given a set of XML inputs, ISLearn can easily learn the constraint in Listing 2.

To avoid overspecialization toward a small set of inputs, ISLearn can automatically *validate* constraint candidates—by generating further inputs from them and checking whether these inputs would be accepted by the program. This also works in *debugging* scenarios: If we have a set of inputs for which a specific property holds (say, the length of some input element exceeds some constant), ISLearn will not only learn that constraint, but can also ensure that further instantiations of the constraint reproduce the failure.

2.5 Summary

With these examples, we have demonstrated how ISLa constraints precisely characterize input classes associated to some program behavior. Developers can use these descriptions to obtain *semantically valid* inputs, *describe* the conditions of discovered bugs, and for targeted *triggering* of such bugs. Given existing inputs, ISLearn can determine constraints that precisely characterize input properties and program behavior.

Note that *without* ISLa and ISLearn, implementing any of these constraints can be a tiresome experience. While a handwritten generator can easily ensure matching XML tags or usage of tags from a dictionary, proper handling of namespaces is already a challenge and solving arithmetic constraints over multiple elements will be increasingly difficult. Extending such a generator to be composable *and* usable as a parser for checking or mutating inputs will require an effort comparable to implementing most of ISLa, but the resulting tool will not be nearly as versatile.

3 ISLA SYNTAX AND SEMANTICS

ISLa constraints are built from a *signature* of grammar, predicate, and variable symbols. We first formally define CFGs, following [15, Chapter 5]; afterward, we introduce ISLa signatures.

Definition 3.1 (Context-Free Grammar). A Context-Free Grammar (CFG) is a tuple $G = (N, T, P, S)$ of (1) a set of *nonterminal symbols* N, (2) a set of *terminal symbols* T disjoint from N, (3) a set of *productions* P mapping nonterminals $n \in N$ to an *(expansion) alternative*. An alternative is a string of terminal or nonterminal symbols. Formally, $P \subseteq N \times (N \cup T)^*$; and (4) a designated *start symbol* $S \in N$.

By convention, we surround nonterminal symbols with angular brackets (e.g., $\langle start \rangle$). Signatures contain a special nonterminal symbol "int" for *numeric* variables representing derivation trees whose string representations correspond to a natural number.

Definition 3.2 (ISLa Signature). A *signature* is a tuple $\Sigma = (G, \text{PSym}, \text{VSym})$ of a *grammar* $G = (N, T, P, S)$, a set of *predicate symbols* PSym of strictly positive *arity*, and a set of typed variable symbols VSym. The type $vtype(v)$ of $v \in$ VSym is a symbol $n \in N \cup \{\text{int}\}$, $\text{int} \notin N$.

We now define the syntax of Core-ISLa formulas. We assume underspecified sets $\text{Trm}_{bool}(vars)$ of Boolean SMT-LIB terms with free variables *vars*. These sets contain the constants `true` and `false`, and S-expressions `(f` a_1 `...` a_n`)`, where `f` is an n-ary function symbol of *Bool* sort and the a_i are SMT expressions of suitable sort. Formulas in $\text{Trm}_{bool}(vars)$ may contain uninterpreted string constants whose names coincide with the names in *vars*. For the precise definition of SMT-LIB terms, we refer to the SMT-LIB standard [1] and the repository of SMT-LIB theories [28]. Apart from SMT-LIB expressions, quantifiers, and Boolean combinators, ISLa uses *predicate formulas* with predicate symbols from PSym. While our definition of Core-ISLa formulas is parametric in PSym, the ISLa solver comes with a set of predefined predicates such as `inside` from Listing 3. For a list of ISLa's built-in predicates, we refer to our electronic appendix [25] and the ISLa language specification [24].

Definition 3.3 (Core-ISLa Formulas). The set Fml of *ISLa formulas* for a signature $\Sigma = (G, \text{PSym}, \text{VSym})$, with $G = (N, T, P, S)$, is inductively defined as:

(1) $\varphi \in$ Fml if $\varphi \in \text{Trm}_{bool}(\text{VSym})$.
(2) $p(v_1, \ldots, v_n) \in$ Fml for each predicate symbol $p \in$ PSym with arity n and $v_i \in$ VSym.
(3) (**not** φ), (φ **and** ψ), (φ **or** ψ) are in Fml for $\varphi, \psi \in$ Fml.
(4) **forall** *type* x **in** y: φ and **exists** *type* x **in** y: φ are in Fml for $x, y \in$ VSym, $vtype(x) = type \in N \cup \{\text{int}\}$, and $\varphi \in$ Fml.
(5) **forall** *type* x=*"mexp"* **in** y: φ and its existential counterpart **exists** *type* x=*"mexp"* **in** y: φ are in Fml if $x, y \in$ VSym, $vtype(x) = type \in N$, and $\varphi \in$ Fml, and *mexp* is a string consisting of symbols in $N \cup T$, non-nested lists "$[\cdots]$" of such symbols (*optional* symbols), and variables references "$\{t \ v\}$," where $v \in$ VSym and $t = vtype(v)$.

We use "φ **implies** ψ" as a shorthand for "(**not** φ) **or** ψ."

The set Fml is relative to a signature Σ, left implicit for simplicity. Parentheses can be omitted according to the following precedence rules: Quantifiers bind stronger than negation, which binds stronger than conjunction, which binds stronger than disjunction.

We only consider ("top-level") ISLa formulas containing *exactly one* unbound variable, which is the default `start` constant, or the one specified in the optional **const** declaration.

Simplified ISLa. The simplified ISLa language features are defined in terms of a translation to Core-ISLa. We briefly list these features and sketch how they are mapped; for a more detailed discussion, we refer to the ISLa language specification [24].

Generalized SMT-LIB expressions. ISLa allows writing binary SMT-LIB expressions in infix syntax "`x` `op` `y`" and all other expressions in standard mathematical prefix syntax "`f(...)`." They are mapped to SMT-LIB S-expressions.

Omission of "in start." Omitting the "**in** `...`" part in a quantified expression is permitted. It defaults to "**in** `start`," where `start` is replaced by an explicitly specified constant, if any.
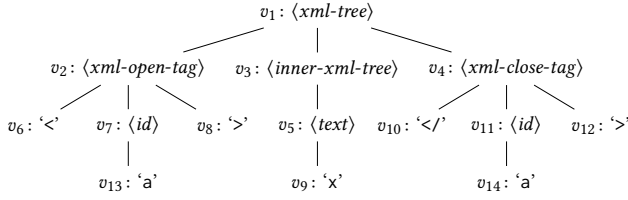
Figure 2: Example XML derivation tree.

Omission of bound variable names. Variable names in quantifiers, such as "var" in "**forall** `<type>` var: ...," are optional. The Core-ISLa translation introduces a fresh variable.

Free nonterminals. Nonterminal types can be used instead of variables in atomic formulas and "**in** ..." expressions. The mapping to Core-ISLa adds a new, top-level **forall** quantifier binding a fresh variable, and replaces occurrences of the nonterminal in the original formula with that variable.

"X-Path" expressions. The notation "parent.`<child>`[i]" (child axis) allows addressing the *i*-th *immediate* `<child>` element of parent; "[i]" is optional and defaults to "[1]." The descendant axis expression "parent..`<child>`" refers to all `<child>` elements in the subtree of parent (i.e., not only immediate children). Child and descendant axis expressions can be chained to longer expressions. They can be used instead of variables in atomic formulas and "**in** ..." expressions. Child axis expressions translate to match expressions; descendant axis expressions to universal quantifiers.

Semantics. The semantics of an ISLa constraint are all strings derivable from the reference grammar that satisfy the constraint. To make this precise, we first define derivation trees. Then, we fix the meaning of ISLa constraints by defining a validation judgment.

Definition 3.4 (Derivation Tree). A *derivation tree* for a CFG $G = (N, T, P, S)$ is a *rooted ordered tree* such that (1) all vertices v are *labeled* with symbols $label(v) \in N \cup T$, where the root is labeled with S, (2) if $v_1, \ldots, v_k$ are the children of a node labeled with n, then there is a production $(n, (s_1, \ldots, s_k)) \in P$ such that for all v_i, $label(v_i) = s_i$. For a derivation tree t, we write $leaves(t)$ for the set of its leaves, and $label(t)$ for the label of its root. A derivation tree is *closed* if $l \in T$ for all $l \in leaves(t)$, and *open* otherwise. $\mathcal{T}(G)$ is the set of all (closed and open) derivation trees for G.

Example 3.5. Fig. 2 visualizes the derivation tree of the XML document "<a>x" for the XML grammar in Fig. 1: The tree's root, v_1, is labeled with the grammar's start symbol $\langle xml\text{-}tree \rangle$; its edges conform to the possible grammar derivations. Consider, e.g., node v_2 and its immediate children v_6, v_7, and v_8. According to Definition 3.4, there has to be a production $(\langle xml\text{-}open\text{-}tag \rangle, ('<', \langle id \rangle, '>'))$ in the grammar, which is indeed the case, since '<' $\langle id \rangle$ '>' is an expansion alternative (the first one) for the nonterminal $\langle xml\text{-}open\text{-}tag \rangle$. The leaves $leaves(t)$ are $\{v_6, v_{13}, v_8, v_9, v_{10}, v_{14}, v_{12}\}$. The tree t is *closed*, since all leaves are labeled with *terminal* symbols. It would be *open* if we removed the subtree rooted in any tree node (but the root).

We convert a derivation tree to a string (written "$str(t)$") by concatenating its leaves in order of their occurrence. If t is the tree from Fig. 2, we have $str(t) =$"<a>x".

Match Expressions. For evaluating ISLa formulas, we have to match quantified formulas with match expressions against derivation trees. To that end, we use a partial function $match(t, mexpr)$ from trees t and match expressions $mexpr$ to mappings from variables to subtrees. We say that *there is a match m* for t and $mexpr$ if *match* returns a such a mapping. Our implementation parses the match expression and recursively matches the result against t.

Validation. We define the semantics of Core-ISLa formulas (the semantics of the simplified ISLa language features follows from their translation to Core-ISLa) by a *validation judgment* $\pi, \sigma, \beta \models \varphi$, where π and σ are interpretations of predicate symbols and SMT expressions, and the variable assignment β is a substitution of derivation trees for variables. The intuition of this judgment is that φ holds (evaluates to *true*) when instantiating free variables in φ according to β under the interpretations of predicates and SMT expressions as provided by π and σ. We write $\beta(\varphi)$ for the substitution of free variables in φ by their assignments in β, and $\beta[v \mapsto t]$ for the updated assignment where the variable v is now mapped to the tree t. For a match $m = match(t, mexpr,)$, we write $\beta[m]$ for $\beta[v_1 \mapsto t_1] \cdots [v_n \mapsto t_n]$, where $v_i \mapsto t_i$ are all assignments in m. The primitive substitution of t for v is denoted by $\{v \mapsto t\}$. By $\beta^{\downarrow}$ we denote the assignment of variables to *strings* instead of trees: If β associates v with t, $\beta^{\downarrow}$ associates v with $str(t)$.

In the definition of the validation judgment, $\top$ and $\bot$ represent semantic truth and falsity, resp. Note that we expect σ to always return $\top$ or $\bot$. Timeouts, not uncommon for SMT solvers, are usually no problem for closed formulas without free variables. Should the solver time out anyway, we interpret this as $\bot$.

Definition 3.6 (ISLa Validation). Let $\Sigma = (G, \text{PSym}, \text{VSym})$ be a signature, $\pi : \text{PSym} \to \mathcal{T}(G)^* \to \{\top, \bot\}$ an interpretation of predicate symbols, $\sigma : \text{Trm}_{bool}(\emptyset) \to \{\top, \bot\}$ an interpretation of closed SMT S-expressions, and β a variable assignment. We inductively define the judgment $\pi, \sigma, \beta \models \varphi$ as

(1) $\pi, \sigma, \beta \models \varphi$ iff $\varphi \in \text{Trm}_{bool}(\emptyset)$ and $\sigma(\beta^{\downarrow}(\varphi)) = \top$.

(2) $\pi, \sigma, \beta \models p(v_1, \ldots, v_n)$ iff $\pi(p)(\beta(v_1), \ldots, \beta(v_n)) = \top$.

(3) $\pi, \sigma, \beta \models$ **not** φ iff not $\pi, \sigma, \beta \models \varphi$.

(4) $\pi, \sigma, \beta \models \varphi$ **and** ψ iff $\pi, \sigma, \beta \models \varphi$ and $\pi, \sigma, \beta \models \psi$.

(5) $\pi, \sigma, \beta \models \varphi$ **or** ψ iff $\pi, \sigma, \beta \models \varphi$ or $\pi, \sigma, \beta \models \psi$.

(6) $\pi, \sigma, \beta \models$ **forall** *type* v **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t] \models \varphi$ holds for all subtrees t in $\beta(w)$ whose root is labeled with *type* $\in N$.

(7) $\pi, \sigma, \beta \models$ **forall int** n **in** φ: iff $\pi, \sigma, \beta[n \mapsto t] \models \varphi$ holds for all trees t such that $str(t)$ represents a number in $\{0, 1, 2, \ldots\}$.

(8) $\pi, \sigma, \beta \models$ **exists** *type* v **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t] \models \varphi$ holds for some subtree t in $\beta(w)$ whose root is labeled with *type* $\in N$.

(9) $\pi, \sigma, \beta \models$ **exists int** n **in** φ: iff $\pi, \sigma, \beta[n \mapsto t] \models \varphi$ holds for some tree t such that $str(t)$ represents a number in $\{0, 1, 2, \ldots\}$.

(10) $\pi, \sigma, \beta \models$ **forall** *type* $v=$"*mexpr*" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for all subtrees t with root r in $\beta(w)$ such that $label(r) = type$ and there is a match $m = match(t, mexpr)$.

(11) $\pi, \sigma, \beta \models$ **exists** *type* $v=$"*mexpr*" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for a subtree t with root r in $\beta(w)$ such that $label(r) = type$ and there is a match $m = match(t, mexpr)$.
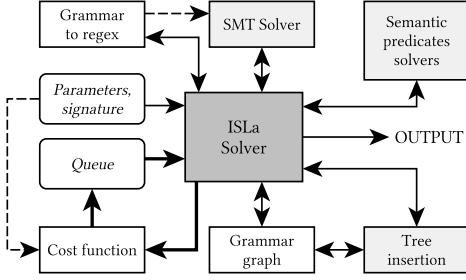
Figure 3: ISLa solver schema. Bold arrow lines depict the main solver loop. Light gray rectangles are main constraint-solving components; the remaining ones are auxiliary components.

Example 3.7. Consider the constraint for well-balanced XML trees from Listing 2, and the XML tree t from Fig. 2 for the document "<a>x." We evaluate whether this tree is well-formed, starting from an initial assignment $\beta = \{\text{start} \mapsto t\}$. Since the outermost element of the constraint is a universal formula with match expression, Item (10) of Definition 3.6 applies. Thus, we have to prove that $\pi, \sigma, \{\text{start} \mapsto t\}[v \mapsto t][m] \models \varphi$ holds for all instantiations, i.e., tree elements with root $\langle xml\text{-}tree \rangle$ matching the match expression (i.e., not a self-closing tag). There is a match m in t, instantiating opid to a and clid to a. Thus, it remains to show that $\sigma((= \text{"a" "a"})) = \top$, which is the case.

Definition 3.8 (ISLa Semantics). Let $\varphi \in \text{Fml}$ be an ISLa formula with the single free variable c for the signature $(G, \text{PSym}, \text{VSym})$, and π, σ be interpretations for predicates and SMT formulas. We define the semantics $[\![\varphi]\!]$ of φ as

$$[\![\varphi]\!] := \{str(t) \mid t \in \mathcal{T}(G) \land closed(t) \land \pi, \sigma, \{c \mapsto t\} \models \varphi\}.$$

4 SOLVING ISLA CONSTRAINTS

Our ISLa solver stepwise expands elements from a queue of *Conditioned Derivation Trees (CDTs)*. A CDT is a pair $\Phi \triangleright t$, where Φ is a set of ISLa formulas and t a—possibly open—derivation tree. Intuitively, the conjunction of the formulas in Φ constrains the inputs represented by t, similarly as $[\![\varphi]\!]$ constrains the language of the grammar. Open trees represent the possibly infinite set of derivation trees that can be derived from them by expansion according to the grammar rules; imposing constraints potentially reduces the set of applicable rules and thus the represented concrete trees. On the other hand, *closed* derivation trees only stand for themselves. If a constraint is added to a closed tree, the result is either empty (if the tree does not satisfy the constraint) or consists of the tree itself.

To enable references to trees in constraints, we assign unique, numeric identifiers to derivation tree nodes. These identifiers may be used instead of *free* variables in ISLa formulas (variables bound by quantifiers may not be replaced with tree identifiers).

Consider, for example, the ISLa constraint

$$\varphi = \textbf{forall } \langle id \rangle \ id \textbf{ in } start: \texttt{str.len}(id) = 3$$

constraining the XML grammar in Fig. 1 to identifiers of length 3. Let t be a tree consisting of a single (root) node with identifier

1, and labeled with $\langle start \rangle$. Then, $[\![\varphi]\!]$ is identical to the strings represented by the CDT

$$\{\textbf{forall } \langle id \rangle \ id \textbf{ in } \textbf{1}: \texttt{str.len}(id) = 3\} \triangleright t.$$

Fig. 3 schematically represents the ISLa constraint solver. We formalized the solver as a CDT *transition system* in our electronic appendix [25]. Starting with the CDT above, the solver expands the open tree t according to the grammar and adds the resulting CDT into the queue. The queue itself is a *priority queue*. The order of CDTs inside the queue is determined by a configurable *cost function*.

Expansion continues as long as it gets us nearer to matching the universal quantifier (in the example, until an $\langle id \rangle$ nonterminal symbol is contained in the trees resulting from the expansion). Eventually, the following state will be added to the queue:

$$\{\textbf{forall } \langle id \rangle \ id \textbf{ in } \textbf{1}: \texttt{str.len}(id) = 3\} \triangleright <\langle id \rangle/>$$

Now, the universal quantifier matches and is instantiated. Let **4** be the identifier of the subtree labeled with the $\langle id \rangle$ nonterminal. Then, we obtain (using bold font for the tree identifier in the SMT formula resulting from the instantiation):

$$\{\texttt{str.len}(\textbf{4}) = 3,$$
$$\textbf{forall } \langle id \rangle \ id \textbf{ in } \textbf{1}: \texttt{str.len}(id) = 3\} \triangleright <\langle id \rangle/>$$

The solver now removes the quantified formula from the constraint set, since there is no chance of obtaining *another* $\langle id \rangle$ by further expansion. Next, it invokes the SMT solver to obtain a solution for the formula $\texttt{str.len}(\textbf{4}) = 3$. If we simply asked the solver for a string of length 3, we would not necessarily receive an answer matching the language of the $\langle id \rangle$ nonterminal; for example, the solver could produce a sequence of three space characters. Thus, we use the "Grammar to regex" component to produce a regular expression describing the desired syntax, which we add to the solver query. While generally, it is not possible to *precisely* transform a CFG into a regular expression, it is often feasible for small sub grammars, like the one for the $\langle id \rangle$ nonterminal. Otherwise, we create an approximate regular expression by unfolding problematic recursions up to a fixed bound.

The solution returned by the solver is parsed into a derivation tree and substituted for the subtree with identifier **4**; the SMT formula is removed from the constraint set. This results in a set of CDTs (the number of solutions requested from the SMT solver is configurable) with empty constraint sets and closed trees such as $\{\} \triangleright <abc/>$. Since there are no constraints and open tree leaves left, $<abc/>$ is immediately output as a solution of the constraint.

The solver not only stops tree expansion if it can be sure that no universal quantifier can eventually be matched by doing so; it also only expands open subtrees for which this is the case. Consequently, there are situations where the constraint set is empty, and the associated derivation tree still open. In that case, *any* expansion of the tree is admissible. The solver then calls a standard grammar fuzzer to close the tree using random expansions (again, the number of requested solutions is configurable). This procedure ensures that the solver does not generate too many solutions that look alike by considering all possible grammar expansions in all cases.

There are two more constraint solver components, which provide solutions for *existential quantifiers* and *semantic predicates*.

Existential Quantifiers. Existential quantifiers (e.g., "there is an outer XML tag defining a given namespace") not matching the current derivation tree are eliminated using the "tree insertion" component, which searches for opportunities to insert the requested tree into the existing constrained derivation tree. The inserted tree contains a node labeled with the nonterminal type of the variable bound by the quantifier, and optionally contains subtrees for match expression elements. For the XML namespace example, the component will, e.g., *replace* an existing ⟨*xml-tree*⟩ subtree with the tree to insert, and in turn *add the replaced tree as a subtree of the inserted tree*. Tree insertion can cause violations of *already eliminated* constraints. Thus, the *original constraint is re-inserted* afterward. If none of the already solved constraints were violated, the added constraint is quickly eliminated again. Alternatively, the added CDT is discarded, or further insertions are performed

Semantic Predicates. ISLa distinguishes *structural* and *semantic* predicates. Structural predicates, like "`inside`" in Listing 3, evaluate to *true* or *false*. *Semantic* predicates, for which specific solvers have to be implemented, can additionally evaluate to an *assignment*, similarly to SMT formulas, or *"not ready"* if the result differs for different expansions of open argument derivation trees. We use them to address shortcomings of SMT solvers or the SMT-LIB language. Classic use cases are *checksum* predicates—encoding checksums in SMT-LIB is cumbersome at least—and structure-aware predicates like the `count` predicate used in our CSV case study, which produces rows with a specific number of columns.

Quantifiers over Integers. Existential *numeric* quantifiers are eliminated by introducing fresh numeric constants. Universal quantifiers are more complicated. If their core restricts the range of the quantified variable, ISLa enumerates all possible values. Additionally, the solver implements *transformations* for formulas of a specific structure. For example, formulas of the shape "**forall int** i: **exists** <A> elem: **not** A(elem, i)" get

```
exists int i: (
  exists <A> elem': phi(elem', i) and
  exists <A> elem: not phi(elem, i))
```

if A(elem, i) holds for exactly one i when elem is fixed. The result can be solved by fresh constant introduction.

Conjunctions, Disjunctions, Negations. The solver pushes negations inside formulas, splits conjunctions into several elements of the constraint set of one CDT, and disjunctions into several CDTs.

Cost Function. The choice of the cost function impacts the solver's performance, both in terms of efficiency (generated inputs per second) and diversity (input features covered). Our cost function computes the weighted geometric mean of different *cost factors*. We provide a sensible default weight vector. Furthermore, weights can be manually configured, and we provide an optimizer using an evolutionary algorithm for choosing good weights. We currently consider five cost factors: (1) *Tree closing cost.* We approximate the cost to close a derivation tree by the sum of the estimated instantiation effort for all leaf nonterminal symbols. (2) *Constraint cost.* This assigns higher cost to constraints that are more expensive to solve, notably existential quantifiers that have to be eliminated by tree insertion. (3) *Derivation depth.* Assigning higher costs to

CDTs generated later in the process can prevent starvation of states added earlier. (4) *k-path coverage.* We use the k-path coverage metric [11] to determine the context-sensitive input feature coverage of derivation trees. We penalize trees covering only few k-paths. The concrete value of k is configurable; the default is 3. (5) *Global k-path coverage.* This factor assigns a higher cost to trees whose k–paths have already been covered by existing trees in the queue. The history of covered paths is reset once all paths have been covered.

5 MINING ISLA CONSTRAINTS

The ISLa components introduced so far enable developers to *manually* specify input constraints based on an analysis of input formats, and use the constraints for input validation and generation. With these constraints, developers do not have to code domain-specific input generators or checkers; furthermore, ISLa constraints can easily be refined and specialized, e.g., by adding another constraint for targeted testing. Yet, the full potential of such a declarative specification language materializes when we *automatically mine* input constraints from samples and automatic experiments. This enables us to *connect any observable program behavior with constraints on system inputs*. Example behaviors of interest include normal completion, reaching some point in code, or crashing.

To that end, we developed ISLearn, a *miner for input constraints*. ISLearn is inspired by Daikon [7], a tool for learning unit-level program invariants from dynamic execution traces. Daikon checks for invariants from a predefined set of *patterns* (e.g., value ranges, sortedness). ISLearn also uses patterns. The main differences to Daikon are: (1) ISLearn mines and combines *quantified*, *structure-aware* formulas; Daikon generates invariants over literals or simple collections, (2) Daikon requires a meaningful *test suite* to obtain feasible unit-level execution traces. ISLearn can *automatically generate more inputs* satisfying a program property, and *reduce* those inputs to their *essential features*, (3) ISLearn also considers *negative* inputs (not satisfying a property) to estimate the *specificity* of invariants, and (4) ISLearn can easily be *extended with more patterns* by adding them to a human-readable configuration file. Manual extensions *may not be required*, though: ISLearn comes with a default catalog populated with general patterns and patterns inspired by our ISLa case studies. All these are unique features of ISLearn.

The main inputs to the ISLearn system, apart from a *grammar* of the input language (possibly mined by other tools [10, 18]), are sets of positive and negative *sample inputs*, and a *program property* (e.g., the program terminates normally). Both are optional: Invariants can be mined from inputs only, and inputs can be automatically generated from only the property.

Patterns are defined in a superset of the ISLa language, enriched with *placeholders* for *nonterminal types* (`<?NONTERMINAL>`), *match expressions* (`<?MATCHEXPR(params)>`, where params is a list of variables that should be bound in the instantiated match expression, and *string constants* (`<?STRING>`). The `<?DSTRINGS>` placeholder can be instantiated by *multiple* strings; the surrounding, atomic ISLa formula is expanded for all instantiations.

Consider the constraint for prefix bindings in XML attributes from Listing 3. We abstract this constraint to an ISLearn pattern by replacing all nonterminal types and match expressions by corresponding placeholders. The constant `"xmlns"` is abstracted by

```
forall <?NONTERMINAL> attribute="{<?MATCHEXPR(prefix_use)>}"
  in start: (not prefix_use = <?DSTRINGS> implies
 exists <?NONTERMINAL> outer_tag="{<?MATCHEXPR(cont_attribute)>}"
    in start: (inside(attribute, outer_tag) and
   exists <?NONTERMINAL> def_attribute=
     "{<?MATCHEXPR(prefix_def)>}" in cont_attribute:
    prefix_use = prefix_def))
```

Listing 4: ISLearn pattern obtained from Listing 3

a `<?DSTRINGS>` placeholder to permit instantiations by multiple keywords. The resulting pattern is shown in Listing 4.

This is not the only possible abstraction. In fact, to recover an equivalent invariant for the original XML constraint, we need to introduce another variable ns_prefix in the match expression for def_attribute for binding the constant xmlns, along the constraint "ns_prefix = `<?STRING>`." However, the pattern in Listing 4 is already useful. In our evaluation (Section 6), we applied ISLearn to languages that did not inform our pattern catalog. One of our evaluation targets is the Racket language from the Lisp family. Since Racket programs are, similarly to XML, tree structures, the first abstraction of the XML pattern can be instantiated to a definition-use invariant for Racket. The `<?DSTRINGS>` placeholder is instantiated by all functions used in the learning samples, such as *, +, and sqrt. The constraint prefix_use = `<?DSTRINGS>` is expanded to multiple equalities, one for each used function.

We explain how ISLearn works along the example of the Graphviz DOT language from our evaluation (Section 6.3). We aim for the invariant that edges in directed graphs ("digraph") are directed ("->"), and edges in undirected graphs ("graph") are undirected ("--"). ISLearn operates in three phases. The *input augmentation phase* generates more input samples satisfying (positive) and violating (negative) the program property using a *grammar fuzzer* and a grammar-, property-, and k-path-aware *mutation fuzzer*. The obtained inputs are optionally reduced afterward. Then, learning samples are selected from the positive inputs, minimizing their size while maximizing total k-path coverage. For DOT, example learning inputs are "graph { a -- b; }" and "digraph { a -> b; }." An example of a *negative* input is "graph { a -> b; }."

The *candidate generation phase* instantiates selected patterns from the catalog based on the given learning inputs in several steps. For DOT, we can instantiate the pattern "String Existence:"

```
exists <?NONTERMINAL> elem in <?NONTERMINAL>:
  elem = <?STRING>
```

For example, the first step instantiates nonterminal placeholders in quantifiers and match expression placeholder arguments. The results after each instantiation phase are *approximately filtered* using an ISLa checker for schematic formulas. The filtering is conservative: Whenever *some* learning input *might* satisfy a partially instantiated pattern, that pattern is retained. Two useful instantiations of "String Existence" for DOT are

"**exists** `<GRAPH>` elem in `<graph>`: elem = "graph"" and
"**exists** `<edgeop>` elem in `<edge_stmt>`: elem = "->"."

Finally, the *filtering and combination phase* combines candidate invariants to *conjunctions of disjunctions* satisfying configurable *target values for recall and specificity*. First, we evaluate for each candidate which of the positive and negative inputs it satisfies. From the two learning inputs shown above, the first one satisfies the first constraint, and the second one the second constraint. Then, we

combine candidates to disjunctions up to a configurable size, such that the percentage of positive inputs satisfying the combination exceeds the recall threshold and the recall estimate is greater than that of both candidates alone. For example, we would combine the candidate instantiations above, such that the result satisfies *both* learning inputs. In the next step, we combine the disjunctions to conjunctions to maximize the amount of negative inputs *not* satisfying the resulting combinations (specificity). For instance, the negative input "graph { a -> b; }" satisfies our newly formed disjunction. Thus, we combine it to a *con*junction with a similarly shaped disjunction, only with "digraph" and "--" instead of "graph" and "->". The result has 100% recall and specificity. ISLearn ranks invariants according to their recall and specificity estimates.

Implementation. The ISLa solver and ISLearn are implemented in Python.[1] We use the Z3 SMT solver and a grammar fuzzer based on the Fuzzing Book [33] for finishing unconstrained trees. We implemented additional libraries for grammar graph operations (e.g., k-paths) and approximating grammars with regular expressions.

6 EVALUATION

To evaluate ISLa and ISLearn, we pose three research questions:

RQ1 To which degree do ISLa constraints contribute to the *efficiency* and *precision* of the input generator? With this question, we evaluate how much *benefit* one gets (in terms of more valid inputs) for how much *cost* (in terms of having to specify ISLa constraints).

RQ2 How *diverse* are inputs generated from ISLa constraints? Here, we want to ensure that ISLa does not *overspecialize* (for instance, by producing only a small set of concrete inputs).

RQ3 What are the recall and specificity of invariants mined by ISLearn? We evaluate how useful the invariants mined by ISLearn, and specifically the default patterns, are to describe the circumstances of *normal* program behavior.

Evaluation Subjects. To evaluate RQ1 and RQ2, we identified frequently occurring context-sensitive language properties: (1) Declaration of identifiers (*def-use*), (2) redefinition—identifiers must not be declared more than once (*redef*), and (3) length or counting properties (*len-cnt*). For specific languages, we addressed (4) well-balanced XML expressions ("Balance"), (5) correct TAR checksums ("Checksum"), and (6) consecutive list numbering ("Numbering").

To cover these properties, we chose input languages of different character: (1) One highly structured (XML) and one more human-readable (reStructuredText (reST)) *markup language*, (2) a *data exchange format* (CSV), (3) a *programming language* (Scriptsize-C), and (4) a *binary format* (TAR). Scriptsize-C extends Tiny-C [8] by explicit variable declarations. For each of these languages, we extracted *grammars* from their specifications; for XML, we extended a pre-existing grammar from the Fuzzing Book [33] with namespace prefixes. We then added ISLa semantic constraints to all of these.

The TAR archive format represents properties of binary inputs; it comes with strict length constraints (block sizes) and requires the computation of a checksum. Checksums are generally out of

---

[1]ISLa and ISLearn are available at https://github.com/rindPHI/isla and https://github.com/rindPHI/islearn. They also are published in the Python Package Index (PyPI), see https://pypi.org/project/isla-solver/ and https://pypi.org/project/islearn/.

Table 1: Overview of evaluation targets and their properties. Properties in italic font are not covered by our specifications.

Language	Test Target	*def-use*	*redef*	*len-cnt*	other
Scriptsize-C	clang	✓	✓	✗	*Nontermination*
					Overflow
XML	xml.etree	✓	✓	✗	Balance
TAR	tar	✓	✓	✓	Checksum
reST	rst2html	✓	✓	✓	Numbering
CSV	csvlint	✗	✗	✓	✗
Racket	racket				
DOT	dot			To be mined in Section 6.3	
ICMP Echo	pythonping				

Table 2: ISLa Efficiency, precision, and input diversity

	Constraints	LOC	Efficiency Inputs/min	Precision Inputs/min (%)	Diversity %k-paths	Length #Chars
C	(none)	—	470	113 (24)	53	2 (1)
	+ def-use	6	963	136 (14)	51	28 (28)
	+ no-redef	4	387	387 (100)	58	26 (27)
XML	(none)	—	809	140 (17)	66	10 (5)
	+ balance	2	452	95 (21)	85	41 (35)
	+ def-use	11	435	60 (14)	78	42 (44)
	+ no-redef	5	126	126 (100)	91	44 (48)
TAR	(none)	—	572	0 (0)	0	0
	+ length	44	20	0 (0)	0	0
	+ checksum	3	18	4 (22)	86	3,700 (4,096)
	+ reference	14	19	19 (100)	87	4,121 (4,096)
reST	(none)	—	258	88 (34)	100	13 (10)
	+ reference	6	492	393 (80)	100	32 (32)
	+ length	7	404	379 (94)	100	33 (33)
	+ numbering	7	547	534 (98)	100	32 (33)
	+ no-redef	4	404	404 (100)	100	31 (31)
CSV	(none)	—	804	490 (61)	100	9 (8)
	+ columns	4	127	127 (100)	100	1,103 (828)

The "Efficiency" column considers *all* produced inputs; "Precision," "Diversity," and "Length" only *valid* (accepted) inputs. "Length" is the mean (median) length of all valid inputs. We evaluated k-path coverage for both k=3 and k=4.

the scope of SMT-LIB, which is why we implemented a dedicated semantic predicate for TAR checksums (15 lines of code).

For ISLearn, we chose three *additional* languages to evaluate how well patterns from our catalog transfer to new application scenarios. Again, we aimed at choosing a diverse range of evaluation targets: (1) a *functional programming language* (Racket), (2) a *graph description language* (DOT), and (3) a *binary format* (ICMP packets).

Table 1 gives an overview of languages, test targets, and properties used in our evaluation. For the ISLearn subjects, we leave the properties open, since the goal is to *discover* their invariants. For ground truth, we chose test targets processing each language.

6.1 RQ1: Precision

ISLa aims to produce more valid inputs, at the effort of specifying input constraints. Since ISLa is closed under conjunction, specifications can be added until a satisfying precision is reached. Table 2 relates the lines of ISLa code for a semantic property and the resulting precision. The "(none)" rows stand for "no constraint" added. Here, we ran the grammar fuzzer ISLa uses to close unconstrained open derivation trees. For each language, the rows below "(none)" show the results of the ISLa generator when adding the specified constraint *on top* of the ones appearing above. The first constraint is the one with the most positive effect on precision; similarly for the others. The "Precision" column shows the number of *valid* inputs generated per minute, with the percentage of valid inputs in parentheses. Only 17% of generated XML inputs are valid without

constraints; 140 valid XML documents are generated per minute. For TAR, not a single input is valid. The "Efficiency" column displays the generation speed irrespectively of validity. With the exception of TAR, we observe that ISLa generates dozens to hundreds of inputs per minute, including a high number of valid ones.[2] All values are obtained from the average of two one-hour runs of the generator.

For every constraint added, we provide its length in lines of Core-ISLa code. For Listing 2 (*balance* in Table 2), the length is 2.

Mostly, precision increases with each additional constraint. A typical example is reST, where the first constraint already increases precision from 34% to 80%. For XML, 18 lines of constraints achieve 100% precision. Without constraints, the 17% of valid inputs are almost exclusively made of ⟨*xml-openclose-tag*⟩ elements only. Of the inputs containing an ⟨*xml-open-tag*⟩ element, only 0.03% are valid.

> *A few ISLa constraints suffice to drastically increase the percentage of valid inputs.*

Interestingly, relative precision *declines* when adding the *def-use* property to XML. This stems from the fact that the solver is now directed toward introducing more attributes with namespace prefixes, which introduces more (invalid) attribute repetitions. The *no-redef* constraint increases precision up to 100%. A similar phenomenon can be observed for Scriptsize-C. Still, for all subjects, a few constraints increase the precision to 100%.

> *ISLa constraints can ensure that **all** inputs are valid.*

We would like to emphasize that 100% precision is not a necessary goal. If 80% valid reST documents are sufficient for testing a reST processor, one may decide to stop adding constraints after *reference*.

The most verbose property is the "length" property for TAR, where each field of the archive has to conform to strict length bounds. Yet, the constraint consists of a conjunction of simple constraints (most of them two lines only). If we do *not* provide length and checksum constraints, we cannot produce even a single valid TAR file.

6.2 RQ2: Diversity

A test generator should produce inputs exercising different language features, by which one can expect to reach different paths in the language processor [11]. Essentially, 100% precision can be reached by always producing the same, small input. To validate that ISLa generates *diverse* and thus *interesting* inputs, we compute their *accumulated k-path coverage* [11], assessing how many paths of length *k* in the grammar are present in a derivation tree. The higher the *k*-path coverage, the higher the diversity.

The "Diversity" column in Table 2 shows the percentage of accumulated *3- and 4-paths* during a one-hour run per all 3/4-paths in the grammar. For example, generating XML documents from the grammar only achieves 66% coverage, while we cover 91% of all 3/4-paths when adding all three constraints. We only count valid inputs accepted by the program under test.

Generally, inputs produced by ISLa have *better diversity* than inputs produced without constraints. Only for Scriptsize-C, there

---

[2]There is much potential in optimizing ISLa for speed; e.g., parallel processing of the solver queue and solving simple formulas such as equations without SMT solver calls.

is a small decrease in diversity after adding the *first* constraint; this is compensated after the addition of the second.

To shed some light on the solver's behavior for CSV and Scriptsize-C, we collected information about input *length*. Length is not a particularly good coverage measure: One can always choose, e.g., long identifiers. However, we observe that, in particular for CSV and C, most inputs generated by the grammar fuzzer are trivial; the most common valid C program generated by the grammar fuzzer is "`;`". In general, the ISLa solver clearly outperforms the grammar fuzzer in terms of the complexity of the generated inputs.

> *ISLa covers the diversity of the underlying grammar.*

6.3 RQ3: ISLearn

We populated the pattern catalog for ISLearn with abstractions of the patterns used for the ISLa evaluation targets. In addition, we added some simple properties about magic constants, most notably "String Existence" from Section 5. In this research question, we assess how well ISLearn can be used to mine invariants describing circumstances of normal program behavior (i.e., whether an input is accepted by the program under test) with these patterns. We are particularly interested in two questions: (1) If an input is *valid* (accepted by the program), what is the probability that the mined invariant classifies the input as such (i.e., the ISLa solver reports that the input satisfies the invariant)? This is captured by the *recall* of the invariant. (2) Conversely, if an input is *in*valid, what are the chances that the mined invariant classifies it accordingly? This is assessed by the *specificity* of the invariant.

To evaluate recall and specificity, we chose seed sets of *training* and *validation* inputs. For Racket and DOT, we obtained valid Racket and DOT files from GitHub. We separated those inputs into sets of training and validation inputs of equal size. Subsequently, we expanded the training and validation sets to 50 inputs each using both a mutation-based and a grammar fuzzer. Similarly, we collect negative inputs (not accepted by the programs under test) into sets of negative training and validation inputs, each of size 50. For ICMP Echo, our third evaluation target, we generated random, valid echo request and reply packets using the "pythonping" library. To obtain negative samples, we created arbitrary (not necessarily Echo) ICMP packets, 20% of those with an *incorrect* checksum value.

ISLearn already estimates recall and specificity of invariant candidates based on the supplied sample inputs and returns the top-ranked result. We assessed the quality of that invariant using the *validation* sets. If, e.g., an input from the positive validation set does *not* satisfy the invariant, the input is a *false negative* (FN).

Table 3 presents the *confusion matrices* for our evaluation. For DOT, ISLearn discovered the invariant that edges in directed graphs are directed (`->`), and undirected (`--`) for undirected graphs. The invariant is slightly too weak, as it only requires *one* correct edge in each "edge statement," which, however, can contain multiple (right or wrong) edges. In the case of ICMP Echo packets, the system learns that the value of the "type" is 0 (reply) or 8 (response). It wrongly classifies three packets with wrong checksums as valid. Adding a pattern for a semantic predicate computing *internet checksum* achieves 100% specificity. Both of these invariants are obtained from combined instantiations of the "string existence" pattern. We

Table 3: Confusion matrices for RQ3

(a) DOT

Input	True	False	Total
	Classified as		
True	TP = 50	FN = 0	50
False	FP = 8	TN = 42	50
Total	58	42	100

Recall = 100%, Specificity = 84%
Precision = 86%, Accuracy = 92%

(b) ICMP Echo

Input	True	False	Total
	Classified as		
True	TP = 50	FN = 0	50
False	FP = 3	TN = 47	50
Total	53	47	100

Recall = 100%, Specificity = 94%
Precision = 94%, Accuracy = 97%

(c) Racket (XML pattern)

Input	True	False	Total
	Classified as		
True	TP = 36	FN = 14	50
False	FP = 8	TN = 42	50
Total	44	56	100

Recall = 72%, Specificity = 84%
Precision = 82%, Accuracy = 78%

(d) Racket (XML + reST pattern)

Input	True	False	Total
	Classified as		
True	TP = 36	FN = 14	50
False	FP = 5	TN = 45	50
Total	41	59	100

Recall = 72%, Specificity = 90%
Precision = 88%, Accuracy = 81%

already mentioned that a *def-use* invariant for variables in Racket can be obtained from a pattern derived from an XML invariant; this leads to 72% recall and 84% specificity. One missing semantic feature is a *def-use* property for *functions*. We discovered that by weakening the *def-use* pattern obtained from reST, taking into account pre-defined function symbols that have not been defined, we obtain a suitable invariant for this property. The confusion matrix in Table 3d demonstrates that this increases specificity to 90%. The—compared to DOT and ICMP—low recall stems from the fact that not all pre-defined functions appear in the training set.

> *ISLearn mines invariants of high recall and specificity based on patterns for re-occurring input properties.*

6.4 Threats to Validity

We supported our claim that ISLa is a useful specification language by expressing context-sensitive properties of five subject input languages. Whether indeed ISLa is sufficiently expressive and its solver sufficiently precise depends on whether our choice of subjects is representative. There is a potential threat of *overfitting*, i.e., that we designed ISLa and ISLearn to exactly fit the test subjects. We mitigate this threat by choosing diverse languages, i.e., not only programming or markup languages, or binary formats, but a *mixture* of those. Furthermore, we identified and clustered context-sensitive properties of the test subjects. This supports the claim that those are representative and can be transferred to different targets, as does the fact that an XML pattern could be used for Racket.

7 RELATED WORK

Parser Specifications. ISLa provides a framework to specify *input requirements*, or preconditions, of a program. It targets the system level, where inputs are generally strings. *Parser generators* like ANTLR[3] and the pioneer yacc [16] promoted CFGs for specifying complex structured inputs. However, specifications designed for *parsing* inputs are rarely specific enough to also be used for *producing* valid inputs, which is the gap that ISLa fills.

---

[3] https://www.antlr.org/

Attribute Grammars. Attribute grammars [17] associate grammar symbols with synthesized and derived attributes. This allows checking semantic properties; if attributes use a general-purpose programming language, one can express arbitrarily complex semantic properties. The meta-compiler JastAdd [12], for instance, supports imperative specifications in Java; the same holds for ANTLR (Java) and yacc (C). ISLa's mix of quantifiers, structural predicates, and SMT-LIB assertions allows expressing important input properties and can be used for *parsing* and *producing* inputs alike.

Grammar-Based Test Generation. Context-Free Grammars are well suited for syntax-aware test input generation. CSmith [31] and LangFuzz [14] use CFGs as a basis to randomly create syntactically valid C and JavaScript programs, respectively; Grammarinator [13] produces inputs from ANTLR grammars. The underlying grammars are typically handwritten, but can also be *mined* from programs [10] and inputs [18]. ISLa fits between Grammarinator and CSmith: It can produce inputs from *different language models* like Grammarinator, but fulfills *semantic properties* like CSmith. Yet, the probability that Grammarinator will create a valid TAR file from a CFG approaches zero, and CSmith can only generate—well—C files.

Test Generation with Semantic Properties. FormatFuzzer [6] is a fuzzer for binary formats. It is parameterized with *binary templates* as language models. Those resemble C structs, but come with added code for satisfying semantic constraints, including complex expressions, control statements, and functions. These constraints are strictly *local,* though, mainly supporting checksums and length fields for binary formats. Non-local and complex constraints, such as *def-use* properties, have to be programmatically implemented. ISLa's constraints, in contrast, are declarative, can apply to arbitrary elements in the derivation tree, and are easily solved using Z3.

Pan et al. [22] use Higher-Order Attribute Grammars [30] for fuzz testing, providing custom predicates for parse tree manipulation (e.g., length constraints and checksum computation) in a general-purpose programming language. The approach neither supports parsing nor generation from scratch.

Dewey et al. [5] propose to program grammars and constraints in Prolog using its Constraint Logic Programming (CLP) library for language-based test generation. All their predicates are application-specific, which is the exception for ISLa. There is no tool such as ISLearn to infer Prolog programs. ISLa supports all Z3 theories, whereas CLP only supports integer arithmetic. Finally, CLP-based language fuzzers cannot be used for parsing.

Property-Based Testing. Pioneered by QuickCheck [2], Property-Based Testing (PBT) produces *data structures of the host language* to test individual functions against user-defined properties. This allows expressing features in the host programming language, which is not available when working with unstructured system inputs. ProSyT [4] and Luck [19] generate data structures for Erlang and Haskell, resp., separating semantic constraints from data types.

Generally, the concept of *parsing* and *mutating* existing data is not present in PBT. One exception is Zest [21], which leverages program feedback to create syntactically valid input mutants exercising interesting program paths. The central difference between ISLa and all PBT approaches is that ISLa operates at the system level, producing *system* inputs rather than internal data structures.

Mining Invariants. Daikon [7] is the seminal work for extracting *invariant candidates* from program executions—pre- and postconditions as well as data invariants; its pattern matching approach is the inspiration for ISLearn. Recent advances in the field focus on program verification, loop invariants, and the usage of neural networks [32]. Unlike ISLearn, all these approaches operate at the unit level, and cannot generate targeted executions to refine invariants. To the best of our knowledge, ISLearn is the first approach to specify, determine, and refine invariants at the system level.

8 CONCLUSION AND FUTURE WORK

We proposed ISLa, a declarative specification language for context-sensitive constraints of system inputs. In our framework, syntactic language constraints are specified using Context-Free Grammars (CFGs), which are great for parsing, but often too coarse for generating inputs. Context-sensitive refinements are expressed by ISLa constraints, using the vocabulary defined by the CFG. We formally defined ISLa's syntax and semantics and demonstrated that our ISLa solver can be used to generate semantically correct inputs significantly faster than by generating from a CFG alone. Furthermore, we introduced the ISLearn input invariant miner, which automatically produces useful ISLa specifications based on a program property and/or sample inputs.

Besides further refining the ISLa and ISLearn implementation, our future work will focus on the following topics:

Fuzzer integration. ISLa-generated inputs can serve as high-quality *seed inputs* for graybox fuzzers like AFL; ISLa's checkers can quickly filter out invalid generated inputs. Furthermore, ISLa could improve the performance of *hybrid fuzzers* by serving as a language for exchanging constraints between the symbolic execution and fuzzing components. This idea has been put forward by the authors of the Driller [27] paper, but has not yet been put into practice.

Testing strategies. A probabilistic variant of ISLearn could quickly *learn* which input features *correlate* with program behaviors (including failures or specific coverage); this allows for test generation techniques exploring syntax and semantics.

Constraint synthesis. Besides checking *patterns*, techniques from *program synthesis* would have great potential for generating constraints from examples.

Constraints as oracles. As ISLa allows extracting and assessing arbitrary input elements, it can also check *outputs* for constraints. This allows using ISLa constraints as *oracles* (that could also be learned via ISLearn).

Detecting anomalies. Decomposing inputs and outputs provides plenty of syntactical and semantic *features* that can be used for learning commonalities and anomalies; learned correlations can be reinforced by ISLa-generated tests.

DATA AVAILABILITY STATEMENT

Our evaluated ISLa and ISLearn artifacts are publicly available [26]. The current versions of the prototypes can be downloaded from

https://github.com/rindPHI/isla
https://github.com/rindPHI/islearn

REFERENCES

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[2] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[3] James Clark and Steve DeRose. 1999. XML Path Language (XPath), 2.5: Abbreviated Syntax. https://www.w3.org/TR/1999/REC-xpath-19991116/#path-abbrev Accessed: 2022-08-23.

[4] Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. 2019. Property-Based Test Case Generators for Free. In *Tests and Proofs - 13th International Conference, TAP@FM 2019, Porto, Portugal, October 9-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11823)*, Dirk Beyer and Chantal Keller (Eds.). Springer, 186–206. https://doi.org/10.1007/978-3-030-31157-5_12

[5] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *ACM/IEEE International Conference on Automated Software Engineering, ASE 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 725–730. https://doi.org/10.1145/2642937.2642963

[6] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2021. FormatFuzzer: Effective Fuzzing of Binary File Formats. *CoRR* abs/2109.11277 (2021). arXiv:2109.11277 https://arxiv.org/abs/2109.11277

[7] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 1999 International Conference on Software Engineering, (ICSE)*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 213–224. https://doi.org/10.1145/302405.302467

[8] Marc Feeley. 2001. Tiny-C Compiler. https://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c. Accessed: 2021-10-06.

[9] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[10] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FST)*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679

[11] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 189–199. https://doi.org/10.1109/ASE.2019.00027

[12] Görel Hedin and Eva Magnusson. 2001. JastAdd—A Java-Based System for Implementing Front Ends. *Electron. Notes Theor. Comput. Sci.* 44, 2 (2001), 59–78. https://doi.org/10.1016/S1571-0661(04)80920-4

[13] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[14] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation, 3rd Edition.* Addison-Wesley.

[16] Stephen C Johnson. 1979. Yacc: Yet Another Compiler-Compiler. https://www.cs.utexas.edu/users/novak/yaccpaper.htm. Accessed: 2021-11-19.

[17] Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *Proceedings of the International Conference on Attribute Grammars and their Applications (Lecture Notes in Computer Science, Vol. 461)*, Pierre Deransart and Martin Jourdan (Eds.). Springer, 1–12. https://doi.org/10.1007/3-540-53101-7_1

[18] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

[19] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 114–129. https://doi.org/10.1145/3009837.3009868

[20] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

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

[22] Fan Pan, Ying Hou, Zheng Hong, Lifa Wu, and Haiguang Lai. 2013. Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars. *J. Softw.* 8, 3 (2013), 645–651. https://doi.org/10.4304/jsw.8.3.645-651

[23] Alan J. Perlis. 1982. Epigrams on Programming. *ACM SIGPLAN Notices* 17, 9 (1982), 7–13. https://doi.org/10.1145/947955.1083808

[24] Dominic Steinhöfel. 2022. The ISLa Language Specification. https://rindphi.github.io/isla/islaspec/ Accessed: 2022-08-23.

[25] Dominic Steinhöfel and Andreas Zeller. 2022. Electronic Appendix to "Input Invariants". *CoRR* abs/2208.12049 (2022). https://doi.org/10.48550/arXiv.2208.12049 arXiv:2208.12049

[26] Dominic Steinhöfel and Andreas Zeller. 2022. Replication Package for "Input Invariants". https://doi.org/10.1145/3554336

[27] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

[28] The SMT-LIB Initiative. 2021. SMT-LIB Theories. http://smtlib.cs.uiowa.edu/theories.shtml. Accessed: 2021-10-19.

[29] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. 2020. Theory Strings (SMT-LIB Version 2.6). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml. Accessed: 2021-10-07.

[30] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Richard L. Wexelblat (Ed.). ACM, 131–145. https://doi.org/10.1145/73141.74830

[31] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[32] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 106–120. https://doi.org/10.1145/3385412.3385986

[33] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. Grammar Coverage. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. Accessed: 2021-11-13.