Gregor Engels, Regina Hebig, Matthias Tichy (Hrsg.): SE 2023,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2023 113

Input Invariants

Dominic Steinhöfel[1], Andreas Zeller[2]

Abstract: To exhaustively test a program, we need inputs that the program does not reject. Such *valid* inputs must satisfy syntactic and semantic constraints of the input language. Grammar-based fuzzers efficiently produce *syntactically* valid system inputs but miss context-sensitive *semantic* constraints. Example semantic properties are length fields or checksums in binary inputs or definition-use constraints for variables in programming languages. We introduce ISLa [SZ22a], a *declarative specification language for context-sensitive properties* of structured system inputs. An ISLa specification, or *input invariant*, consists of a context-free *grammar* and a potentially context-sensitive ISLa *constraint*. Our ISLa *fuzzer* produces streams of inputs from invariants. We show that a few ISLa constraints suffice to generate diverse and 100% semantically valid inputs. Additionally, the fuzzer can *repair* and—preserving semantics—*mutate* inputs. Provided sample inputs, a program property, or both, our *ISLearn* prototype *mines* precise invariants. In follow-up work, we used ISLearn for *diagnosing failures*: "The *heartbleed* vulnerability is triggered if `length` exceeds the length of `payload`."

Keywords: fuzzing; specification language; grammars; constraint mining

1 Introduction

To improve a faulty program, one needs to (1) *test* the program to discover—and reproduce—a bug, (2) *debug* it to isolate the problem, and (3) *repair* the fault. The outcomes of step (1) is a bug-triggering *input*. Step (2) provides a sound *hypothesis*; and step (3) a *validated fix*. All these steps require understanding the language of program inputs and, ideally, its outputs. This understanding enables us to exhaustively test a program, evolve a failure hypothesis as a property of program inputs, and thoroughly validate a fix.

Our ISLa specification language can be used to describe a *theory of system inputs*. This theory facilitates *automating* the steps outlined above. ISLa constraints are *declarative*, human-readable, and can be used for testing, program understanding, and failure diagnosis.

2 Specifying Input Invariants with ISLa

An ISLa specification consists of a grammar and constraints. For example, the grammar for the TLS heartbeat extension allows us to decompose a heartbeat request into the sequence "`0x1 <length> <payload> <padding>`." Using an ISLa constraint, we can specify that

[1] CISPA Helmholtz Center for Information Security, Saarbrücken dominic.steinhoefel@cispa.de
[2] CISPA Helmholtz Center for Information Security, Saarbrücken zeller@cispa.de

<length> determines the length of <payload>: "uint16(<length>) = len(<payload>)." When specifying constraints in ISLa, one can use all function symbols in the SMT-LIB theory catalog, in particular from the theory of strings. In addition, ISLa supports *structural* relations (e.g., "before," "inside") and can be *extended* with domain-specific semantic predicates. Two types of quantifiers enable the *precise selection of input elements*. All these constituents result in an expressive yet tractable, grammar-aware string constraint language.

Our ISLa fuzzer generates system inputs by heuristically exploring the input language space, using modern SMT solvers (e.g., Z3) and custom strategies (e.g., for quantifiers) to solve constraints. Our evaluation, based on input languages ranging from C over XML to TAR, demonstrates that the fuzzer efficiently generates *100% precise inputs* from a few lines of ISLa specs while *exercising more language features* than a coverage-based grammar fuzzer.

3 Learning Input Specifications

Writing good specifications takes time, although the effort is well invested: *Code* changes frequently, but *system input specs* (e.g., of protocols or file formats) stay stable for *decades*. Luckily, tools like Mimid [GMZ20] mine *grammars* from parsers. Our ISLearn tool learns semantic constraints from sample inputs, a program property, or both. It instantiates *patterns* obtained from our ISLa case studies and enriched by simple string properties to find input invariants. It derived semantic properties for Graphviz DOT, ICMP Echo, and Racket with an accuracy of 78% to 97%, demonstrating that our patterns apply to new scenarios. In follow-up work, we extended this approach to find diagnoses of program failures for debugging. We integrated *machine learning* to restrict the search to relevant language features, tremendously improving performance; furthermore, an automated *feedback loop* refines candidate hypotheses. Our experimental results on real bug reports show that the generated diagnoses are close to those provided by human developers and that the enhanced tool outperforms both ISLearn and a previously proposed automated debugging tool.

Data Availability

Our ISLa and ISLearn artifacts are publicly available [SZ22b]. The current versions of the prototypes can be downloaded from `https://github.com/rindPHI/{isla|islearn}`.

References

[GMZ20] Gopinath, R.; Mathis, B.; Zeller, A.: Mining Input Grammars from Dynamic Control Flow. In: ESEC/FSE 2020. ACM, 2020.

[SZ22a] Steinhöfel, D.; Zeller, A.: Input Invariants. In: ESEC/FSE 2022. ACM, 2022.

[SZ22b] Steinhöfel, D.; Zeller, A.: Replication Package for "Input Invariants", `https://doi.org/10.1145/3554336`, ACM, 2022.