SISL: Concolic Testing of Structured Binary Input Formats via Partial Specification*

Sören Tempel[1][0000−0002−3076−893X], Vladimir Herdt[1,2][0000−0002−4481−057X], and Rolf Drechsler[1,2][0000−0002−9872−1740]

[1] Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{tempel,vherdt,drechsler}@uni-bremen.de

Abstract. Automatically generating test inputs for input handling routines which implement highly structured input formats is challenging. Existing input generation approaches (e.g. fuzzing) address this problem by requiring verification engineers to create input specifications based on which new inputs are generated. However, depending on the input format, creating such input specifications can be cumbersome and error-prone. We propose simplifying the creation of input specifications by allowing input formats to be only partially specified. This is achieved by utilizing concolic testing (a combination of concrete random testing and symbolic execution) as an input generation technique and thereby allowing parts of the input format to remain unspecified (i.e. unconstrained) symbolic values. For this purpose, we present SISL, a domain-specific language for creating partial input specifications for structured binary input formats.

Keywords: Concolic Testing · Software Verification · Network Protocols.

1 Introduction

Input handling routines are a known source of potentially exploitable bugs in existing software [4]. An emerging dynamic testing technique to uncover these sorts of bugs is concolic testing, a combination of concrete random testing (i.e. fuzzing) and symbolic execution. Employment of concolic testing is limited by the fact that input handling routines often expect inputs to satisfy a complex predefined structured input format (e.g. JSON). As such, invalid inputs are rejected early by the software without performing interesting input processing. Since concolic testing is largely performed with a given time budget, critical bugs remain unnoticed if deeper parts of the software are not reached within that budget.

* This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001.

The outlined problem is of central importance in the fuzzing context. Contrary to symbolic execution, fuzzing performs no formal reasoning and instead relies solely on randomly generated values to create test inputs, thus requiring even more time to satisfy complex input formats. Prior work on fuzzing attempts to address this problem by randomizing individual rules of a specified grammar [1,9,6] or individual fields of specified input blocks [7,3]. Due to the lack of formal reasoning, it is necessary in both cases to manually provide a detailed description of the utilized input format, which can be cumbersome and error-prone. Errors in the provided input specification will cause the software to consider generated inputs as invalid. We propose using concolic testing (which combines fuzzing and symbolic execution) to ease the creation of input format specifications by allowing verification engineers to only partially specify the targeted structured input format. That is, unspecified parts of the input format can be treated as unconstrained symbolic values, thereby allowing an SMT solver—used in symbolic execution for formal reasoning—to automatically fill in the leftover gaps based on extracted program constraints.

We present SISL, a *Domain-Specific Language* (DSL) to partially specify structured binary input formats which are often used in security critical domains (such as the Internet of Things). Furthermore, we illustrate that our proposed language can be easily integrated into existing concolic testing frameworks by proposing an exemplary integration for SymEx-VP [8], a concolic testing engine for embedded RISC-V software. Lastly, we evaluate our DSL by providing evidence that the minimal effort, required to create partial specifications, is outweighed by the gain in coverage and that our proposed DSL is expressive enough to describe a wide range of structured binary input formats. To the best of our knowledge, SISL is the first input format specification language designed explicitly for concolic testing. The SISL tooling is open source and can be obtained from https://agra-uni-bremen.github.io/sisl/.

2 Scheme-based Input Specification Language

The *Scheme-based Input Specification Language* (SISL) is a DSL for partially specifying parametrisable binary input formats for concolic software testing. As the name suggests, SISL is based on the Scheme programming language which, in turn, is a Lisp dialect. We choose Scheme as the basis for our language since it supports hygienic macros which allow defining custom syntactic constructs within the language framework, thereby easing the creation of DSLs [2].

Similar to block-based fuzzers [3,7], SISL allows specifying binary input formats as a sequence of variable-width bit blocks. Contrary to existing work on fuzzing, SISL targets concolic testing and therefore supports distinct block types to distinguish concrete and symbolic values in the specified input format. Symbolic field values can optionally be constrained with symbolic expressions, hence allowing expressing the relationship between different symbolic fields (e.g. $X < Y$ must hold for two symbolic fields X and Y). Unconstrained symbolic fields can be used to leave parts of the input format unspecified, therefore allowing the

```
1  (define-input-format (ipv6-packet next-hdr &encapsulate payload)
2    (make-uint 'version-field 4 ipv6-version-value)
3    (make-uint 'traffic-class 8 0)
4    (make-uint 'flow-label 20 0)
5    (make-uint 'payload-length 16 (input-format-bytesize payload))
6    (make-uint 'next-header 8 next-hdr)
7    (make-uint 'hop-limit 8 42)
8    (make-symbolic 'src-addr 128)
9    (make-symbolic 'dst-addr 128))
10
11 (define-input-format (icmpv6-packet &encapsulate body)
12   (make-symbolic 'type 8 `((Or
13                              (Eq type ,icmpv6-nbr-sol)
14                              (Eq type ,icmpv6-nbr-adv))))
15   (make-symbolic 'code 8)
16   (make-symbolic 'checksum 16))
17
18 (write-format
19   (ipv6-packet
20     icmpv6-next-header
21     (icmpv6-packet
22       (make-input-format
23         (make-symbolic 'body (bytes->bits 32))))))
```

Fig. 1. Excerpt of an example SISL input specification for the ICMPv6 message format.

concolic testing engine to fill in these gaps based on program execution and thus easing the creation of input format specifications. Defined input formats can also be nested, e.g. to express encapsulation in the network protocol context.

An example SISL input specification is provided in Figure 1 were a specification for the ICMPv6 message format is presented. ICMPv6 is a binary network protocol implemented on top of IPv6. For this reason, the SISL specification in Figure 1 defines two input formats. First, the IPv6 message format is defined in Line 1 - Line 9 using SISL's `define-input-format` keyword. This keyword defines a new input format and requires specifying the input format name, optional input format parameters, and the input format fields. In Line 1 the input format name is given as `ipv6-packet`, an optional `next-hdr` parameter is defined, and the special `&encapsulate` keyword is used to denote that the format encapsulates an additional `payload` format. In Line 2 - Line 9 the fields of the IPv6 packet format are defined. Each field definition takes at least two parameters: A field name (expressed as a Scheme symbol) and a field size in bits. Fields can either be concrete or symbolic. Concrete fields require the field value as a third argument. Symbolic fields support an optional third parameter to express symbolic constraints. For `ipv6-packet`, six concrete fields are defined in Line 2 - Line 7. The majority of these fields (Line 3, Line 4, Line 6, and Line 7) use an integer literal as field value. The `version` field (Line 2) uses a predefined variable as a field value and the value of the `payload-length` field depends on the byte size of the `payload` parameter. Furthermore, the `ipv6-packet` definition also uses two symbolic fields for the source and destination address of the IPv6 header format (Line 8 - Line 9). IPv6 addresses have a complex internal structure which is cumbersome to express, by declaring them as symbolic the

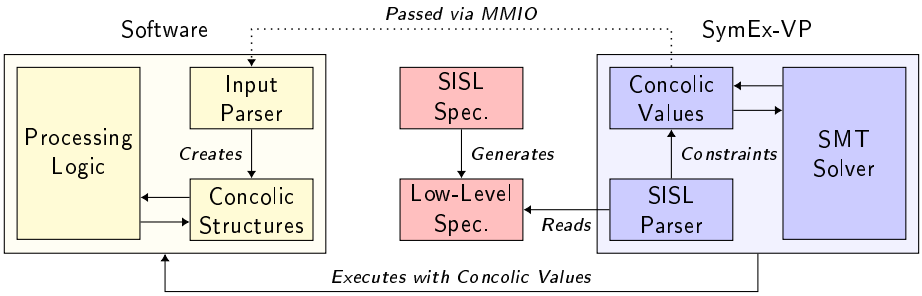

Fig. 2. Overview of our SISL-based concolic testing setup using SymEx-VP.

correct value for these fields will be inferred by the concolic testing engine during execution.

The second input format, defined in Figure 1, is the ICMPv6 message format (Line 11 - Line 16) This definition is analog to the `ipv6-packet` definition, with the exception that it only consists of symbolic fields (Line 12 - Line 16). Furthermore, the symbolic `type` field (Line 12 - Line 14) demonstrates the expression of symbolic constraints on a symbolic field values. Symbolic constraints are expressed as a list of KQuery expressions, a textual representation of symbolic constraints from prior work [5]. In Line 12 - Line 16, the `type` field is constrained so that it either has the value of the variable `icmpv6-nbr-sol` or `icmpv6-nbr-adv`. These two variables refer to constants from the IPv6 *Neighbor Discovery Protocol* (NDP) specification, thereby enabling targeted concolic testing of an NDP implementation with this SISL specification.

To enable such tests, the two described input formats are instantiated in Line 18 - Line 23 of Figure 1 with specific parameters. In this case, the `next-hdr` of the `ipv6-packet` is instantiated with the value of the variable `icmpv6-next-header` and the `payload` parameter is set to an instance of an `icmpv6-packet` which itself has its `body` parameter set to an input format with 32 unconstrained symbolic bytes.

3 Overview and Implementation

We have integrated our proposed DSL with SymEx-VP, an existing open source concolic testing engine for RISC-V embedded software [8]. An overview of the interaction between SISL, the tested software, and SymEx-VP is provided in Figure 2. The central component of Figure 2 is the high-level SISL specification. As discussed in Section 3, this specification is created manually by a verification engineer. Based on the human-readable SISL input specification, a machine-readable low-level specification is automatically generated. This low-level specification is then provided to and read by SymEx-VP which constrains utilized concolic values according to the specification. Since SymEx-VP targets embedded RISC-V software in binary form, the constrained concolic input values are

Table 1. Comparison of concolic testing with SISL and the original SymEx-VP.

Application		SISL			SymEx-VP	
Name	ALOC	SLOC	#Paths	ST	#Paths	ST
Zephyr-CoAP	25383	24	23411	226 min	22999	232 min
Zephyr-IPv6-NDP	31066	30	15122	338 min	1736	453 min
Zephyr-MDNS	31238	35	19585	242 min	2287	452 min

passed to the executed software via *Memory-Mapped I/O* (MMIO) peripheral interfaces (e.g. via a network peripheral) [8]. The software binary is then explored by SymEx-VP based on these input values. Figure 2 (left side) shows a schematic representation of relevant software components performing input processing. Conceptually, the input parser of the software will process the concolic inputs and create data structures based on them. Since the inputs are concolic, the created data structures will also contain concolic values. Based on these concolic values, execution paths through both the input parser and the software processing logic (which processes data structures created by the input parser component) will be enumerated by SymEx-VP. For this purpose, SymEx-VP employs a standard *Dynamic Symbolic Execution* (DSE) concolic testing technique where branches in the software are tracked and negated by an SMT solver to discover new assignments for concolic input values.

By constraining concolic input values prior to execution using SISL, we can (a) reduce the amount of generated input values which are rejected by the software's input parser early on and do not reach the processing logic and (b) reduce the amount of time spend in the SMT solver by using partially instead of fully symbolic inputs, thus reducing the complexity of SMT queries.

4 Experiments and Conclusion

We evaluate our proposed input specification language by applying it to Zephyr[3]. Zephyr is a popular operating system for programming constrained embedded devices in the Internet of Things. For this reason, Zephyr provides input handling routines for structured binary input formats used by different network protocols in this domain. We performed experiments with three different protocol message formats (CoAP, IPv6 NDP, MDNS) using example Zephyr applications. Generated input values were passed directly to the Zephyr network stack through a network peripheral provided by SymEx-VP. The results of our experiments are show in Table 1. For each application, we list the amount of RISC-V assembler instructions (ALOC) in the binary and the amount of SISL lines (SLOC), required for the created input format specification, as a complexity metric. We executed each application for 8 h using the created input specification with our SISL enhanced version of SymEx-VP and with the original SymEx-VP (i.e.

[3] https://zephyrproject.org/

entirely unconstrained symbolic input). For both executions, we list the amount of discovered paths through the program (as a coverage metric, column: #Paths) and the amount of time spend solving constraints on symbolic values (a known bottleneck of concolic testing, column: ST).

The results in Table 1 demonstrate that partial SISL input specifications significantly reduce the amount of solver time, thereby allowing the discovery of more execution paths through a given program in a given time span. The gain in path coverage increases with application complexity (as measured in assembler instructions, column: ALOC). We deem the effort required to create partial input specifications to be comparatively low since complex parts of the input format can be marked as unconstrained symbolic and will thus be inferred during execution. For example, even for a complex input format like MDNS (which is encapsulated in an IPv6 and UDP packet) only 35 lines of SISL specification were required. The utilized SISL specifications and Zephyr applications are available as part of the publication artifacts[4].

In conclusion, we have presented an open source DSL for partial specification of binary input formats in the concolic testing context. Our experiments with Zephyr indicate that our DSL is expressive enough to support different binary input formats and the manual labor required to employ our DSL is outweighed by the benefits in terms of increase in path coverage.

References

1. Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.R., Teuchert, D.: NAUTILUS: Fishing for Deep Bugs with Grammars. In: The Network and Distributed System Security Symposium 2019. NDSS, San Diego, California (Feb 2019)
2. Ballantyne, M., King, A., Felleisen, M.: Macros for domain-specific languages. Proc. ACM Program. Lang. **4**(OOPSLA) (Nov 2020)
3. Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., Vigna, G.: SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In: Information Security. pp. 343–358. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
4. Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., Shubina, A.: Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation. Usenix ;login: **36**, 13–21 (2011)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. p. 209–224. OSDI'08, USENIX Association, USA (2008)
6. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. p. 206–215. PLDI '08, Association for Computing Machinery (2008)
7. Pham, V.T., Böhme, M., Santosa, A.E., Căciulescu, A.R., Roychoudhury, A.: Smart Greybox Fuzzing. IEEE Transactions on Software Engineering **47**(9) (2021)
8. Tempel, S., Herdt, V., Drechsler, R.: SymEx-VP: An open source virtual prototype for OS-agnostic concolic testing of IoT firmware. Journal of Systems Architecture (2022)
9. Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: Grammar-Aware Greybox Fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (2019)

[4] https://doi.org/10.5281/zenodo.6802198