Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers

Aina Niemetz(✉), Mathias Preiner, and Clark Barrett

Stanford University, Stanford, USA
{niemetz,preiner,barrett}@cs.stanford.edu

Abstract. SMT solvers are highly complex pieces of software with performance, robustness, and correctness as key requirements. Complementing traditional testing techniques for these solvers with randomized stress testing has been shown to be quite effective. Recent work has showcased the value of input fuzzing for finding issues, but this approach typically does not comprehensively test a solver's API. Previous work on model-based API fuzzing was tailored to a single solver and a small subset of SMT-LIB. We present Murxla, a comprehensive, modular, and highly extensible model-based API fuzzer for SMT solvers. Murxla randomly generates valid sequences of solver API calls based on a customizable API model, with full support for the semantics and features of SMT-LIB. It is solver-agnostic but extensible to allow for solver-specific testing and supports option fuzzing, cross-checking with other solvers, translation to SMT-LIBv2, and SMT-LIBv2 input fuzzing. Our evaluation confirms its efficacy in finding issues in multiple state-of-the-art SMT solvers.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers determine the satisfiability of formulas over first-order theories and their combinations. They serve as back-end reasoning engines for a wide range of applications in academia and industry [18, 27], including hardware and software verification [14,29,31,35,38,40], model checking [23,24,46], security [12,33], automated test-case generation [22,50], and synthesis [10,34]. Notable SMT solvers include Bitwuzla [42], Boolector [46], cvc5 [13], MathSAT [26], OpenSMT2 [36], SMTInterpol [25], SMT-RAT [28], STP [32], veriT [20], Yices2 [30], and Z3 [41]. State-of-the-art SMT solvers are complex pieces of software with up to hundreds of thousands lines of code. Because of their frequent use as back-ends in higher-level tool chains, strong requirements include performance, robustness, and a high level of trust. Due to their complex nature, full verification of SMT solvers has so far remained out of reach. Furthermore, most SMT solvers are under active development, meaning that there is a constant risk of introducing new issues. While traditional testing techniques

This work was supported in part by DARPA (award no. FA8650-18-2-7861), ONR (award no. N68335-17-C-0558), and by an Amazon Research Award.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 92–106, 2022.
https://doi.org/10.1007/978-3-031-13188-2_5

such as unit testing and a regression test suite are important, these techniques alone are insufficient for achieving high levels of robustness.

SMT solvers usually provide two user-facing interfaces: (i) a textual interface (expecting input in either SMT-LIBv2 [15] or some solver-specific format); and (ii) the application programming interface (API), which allows users to directly integrate the solver into a tool chain. Randomized stress testing (fuzz testing) can be used as a complement to traditional testing to attack these interfaces and has been shown to be very effective at finding issues and thereby helping to improve the correctness and robustness of SMT solvers. In 2009, Brummayer et al. [21] presented a grammar-based generative black-box input fuzzer for the SMT-LIBv1 language [48] called FuzzSMT, and in 2017, Niemetz et al. [45] presented a model-based API fuzz testing framework called BtorMBT for the SMT solver Boolector. More recently, fuzz testing of SMT solvers via their textual interface has gained even more traction with a series of papers on the subject in top venues [19,39,47,49,51,52]. Note that these approaches (and this paper) assume full knowledge of the input structure, i.e., they only generate valid textual input or sequences of API calls. Fuzz testing approaches that are unaware of the input structure can also be useful for testing whether invalid inputs or API calls are handled correctly. This is, however, not a direction we address in this paper.

As mentioned, recent work has focused on fuzzing the textual interface. This is not surprising, as it typically requires significantly less effort than API fuzzing. Input fuzzers generate a new input file or mutate an existing (so-called) seed input file and pass it to a solver binary. Fuzz testing of the solver API is more involved since it requires interaction with the solver—API fuzzers generate sequences of calls to the solver API and typically link against the solver library.

There are, however unique advantages that API fuzzers have. For example, API call sequences generated by API fuzzers may include features and extensions that are not supported by or cannot be expressed via the textual interface. Moreover, even if restricted to standard features, API fuzzers may be able to generate sequences of calls that are not possible using the textual interface, even if the textual interface is built on top of the user-facing API, and especially if it is not. On the other hand, API fuzzing cannot find bugs in parser code. Thus, both fuzzing strategies have unique benefits.

API fuzzing has been an integral part of the development workflow of the SMT solver Boolector [46] since 2013. Boolector supports quantified bit-vector formulas and quantifier-free formulas in the theories of fixed-size bit-vectors, arrays and uninterpreted functions. It ships with BtorMBT [45], an API fuzzer tailored to Boolector, which covers all features of Boolector except quantifiers. BtorMBT has been regularly and rigorously applied during active development of Boolector (locally, prior to major commits to master, and in a cluster setting on 30 nodes prior to every release), with great success. Notably, recent SMT fuzzing campaigns did not report any issues in code covered by BtorMBT [39]; in particular, the few that have been reported [2] all made use of quantified formulas, which are unsupported by BtorMBT. To the best of our knowledge,[1]

[1] The first two authors of this paper are the main developers of the SMT solvers Bitwuzla [42] and Boolector [46], and all three authors are part of the development team of the SMT solver CVC4 [16] and its successor cvc5 [13].

Boolector is the only SMT solver for which API fuzzing has been integrated as a core component of the development workflow.

One of BtorMBT's major weaknesses, however, is that it cannot (easily be extended to) be used with other SMT solvers—it is monolithic, tailored towards the supported theories, and directly calls Boolector's API. Further, it lacks support for quantified formulas, only supports a subset of the theories standardized in SMT-LIB, and even for those, not the full feature set since Boolector only supports a subset. For recording API call sequences, it relies on the API tracing feature of Boolector, the system under test. And for replaying and minimizing such recorded sequences, it requires additional tools.

Contributions. In this paper, we present Murxla, a modular and highly extensible model-based API fuzzer for SMT solvers. Murxla is a comprehensive fuzzing tool that generates valid sequences of solver API calls, records these sequences in a simple text-based trace format, and provides support for minimizing and replaying these traces while preserving the original behavior of the solver. Murxla builds on top of a generic solver interface that can be used with any SMT solver and provides full SMT-LIB support in terms of semantics, features, and standard theories. It further has experimental support for some non-standard theories (sequences, sets, bags) and is fully compatible with and configurable for solver-specific features, extensions, and restrictions. Murxla provides support for option fuzzing (randomly configuring solver options based on the options model of the solver) and can be run in cross-checking mode, where the answers of two different solvers are compared with each other. It additionally implements correctness checks for retrieved model values, unsat assumptions, and unsat cores. Finally, it can optionally translate generated API traces to SMT-LIBv2 (provided that the traces do not contain solver-specific extensions), and can thus be used as a textual interface fuzzer for any solver that supports SMT-LIBv2.

Murxla currently supports the SMT solvers Bitwuzla [42], Boolector [46], cvc5 [13], and Yices2 [30]. Our goal so far has been to fully cover solvers we are actively developing (the first three). We additionally added support for Yices2 as a proof of concept for showing that the tool is sufficiently general and modular to be used with solvers other than our own.

Related Work. The first application of model-based API fuzzing in the context of verification back-ends was proposed by Artho et al. [11] for the SAT solver Lingeling [17]. In the context of SMT solvers, the first and only integration of model-based API fuzzing as a core component of the development workflow was for the solver Boolector [45], as described above. In both instances, the authors showed the effectiveness of the approach for testing solvers, in particular in combination with option fuzzing and delta debugging.

The first input fuzzer for the SMT-LIB language was FuzzSMT [21], a generative grammar-based fuzzer supporting most of SMT-LIBv1 [48]. In 2018, Blotsky et al. [19] presented an SMT-LIBv2 input fuzzer specifically for strings, which generates and mutates SMT-LIBv2 input and mainly targets performance issues. In 2020, Winterer et al. [51,52] proposed two mutational approaches, one based on merging two inputs and the other based on mutating operators. The former supports only integers, reals, and strings, whereas the latter supports all

benchmarks in SMT-LIB but only mutations for the most basic operators. In the same year, Mansur et al. [39] presented Storm, an SMT-LIBv2 fuzzer based on mutating the Boolean structure of an input. Most recently, Park et al. [47] presented TypeFuzz, a hybrid approach for integers, reals, and strings which mutates SMT-LIBv2 by replacing expressions with newly generated expressions. Finally, Scott et al. [49] recently proposed a mutational fuzzer for all of SMT-LIB which leverages reinforcement learning and targets performance issues.

2 Model-Based API Fuzzing for SMT Solvers

Generally speaking, model-based API fuzzing can be seen as lifting grammar-based input fuzzing to the API level: it requires a "model" of the solver that defines what sequences of API calls are valid. For convenience, we consider this model to be made up of three distinct parts: (i) the *semantic (or data) model*, which defines constructs (such as theories, sorts, operators, and commands) and their semantics (usually based on the SMT-LIBv2 [15] standard); (ii) the *API model*, which defines the usage of the API itself; and (iii) the *options model*, which defines configuration options and how they may or may not be combined.

The main requirements for SMT solvers, especially when used as back-ends of higher-level tool chains, are correctness, performance, and robustness. Within the SMT community, the notion of "issue" is thus commonly defined as one of the following: (i) *soundness* issues—either refutation unsoundness (the solver answers *unsat* when the input is *sat*) or model unsoundness (the solver answers *sat* when the input is *unsat*); (ii) *incorrect witnesses*—models (values), proofs, unsat cores, or unsat assumptions; (iii) *crashes*—assertion failures, segmentation faults; and (iv) *performance regressions*. The most critical issues are soundness issues. Refutation unsoundness is especially problematic, as most solvers provide limited or no means for checking the correctness of an *unsat* result. Model unsoundness is less problematic, since state-of-the-art SMT solvers usually provide models for satisfiable formulas, which are easier to check for correctness. The easiest way to identify soundness issues is to check one solver against a second solver, unless the satisfiability of the input formula is known or can be determined by construction. Witnesses are very often checked inside the solver when in debug mode, and their correctness can be determined outside the solver with relatively little effort for all but proofs, which require more involved checking.

As SMT solver developers, we are interested in catching issues as close to the source as possible. For that purpose, in the context of model-based API fuzzing, we configure solvers under test in debug mode with assertions enabled.

3 Murxla

Murxla is a modular model-based API fuzzing tool for SMT solvers which generates valid solver API call sequences and supports the recording, replaying, and minimizing of these sequences for debugging purposes. Murxla is written in C++ and available under the GPLv3 at [43]. Extensive documentation is

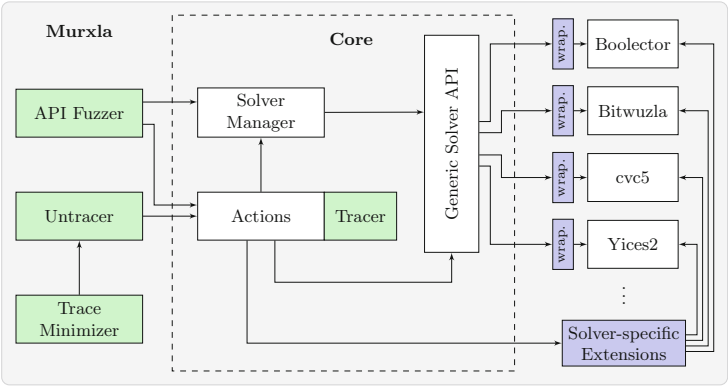

Fig. 1. Murxla architecture. (Color figure online)

available at [44]. A high-level view of its architecture is given as a call graph in Fig. 1. Murxla can integrate any SMT solver (provided that it exposes an API in a programming language that can be integrated). Murxla provides a solver API abstraction, the *Generic Solver API*, which is then specialized via a solver wrapper for a specific solver. Solver-specific components are indicated in blue in Fig. 1 and consist of the solver wrappers and solver-specific extensions of the general API model and options model implemented by Murxla. The four main components of Murxla (green) are the *API Fuzzer*, the *Tracer*, the *Untracer*, and the *Trace Minimizer*.

The API Fuzzer is responsible for generating random but valid API call sequences to the solver under test. The Tracer records these sequences in an *API trace*, which stores all the information required to replay the trace with the Untracer. Replaying a trace with the Untracer executes the exact same API call sequence that was executed when recording the trace. This is particularly useful for replicating interesting behavior that was uncovered while fuzzing the API of the solver under test. The Trace Minimizer takes an API trace as input and tries to minimize it while preserving its behavior with respect to the solver under test. Murxla's core connects all of these components. It is also responsible for interfacing with the SMT solvers and maintaining all sorts and terms created by a solver. In the following, we will describe these components in more detail.

3.1 The Core

Murxla's Core manages communication and the sorts and terms created by a solver. It consists of three modules: the *Actions*, the *Solver Manager*, and the *Generic Solver API*.

Actions. An action is an abstraction defining a particular interaction with the solver under test. These interactions are represented internally as a set of calls to the Generic Solver API. Actions are responsible for three tasks: (1) randomly

generating API call arguments; (2) *executing* API calls with a given set of arguments; and (3) *replaying* a traced copy of the action.

Murxla currently implements a base set of 25 actions which wrap the methods of the Generic Solver API and include creating and deleting a solver instance, configuring solver options, creating sorts and terms, asserting formulas, altering the context levels via push and pop, checking satisfiability of asserted formulas (with assumptions), and many more. When executing API calls, actions may perform sanity checks on results retrieved from solver API calls. For this, Murxla provides a macro `MURXLA_TEST` which allows C-style assertion checks. These remain in the code even if the tool is compiled without assertions. If a solver supports more functionality than that covered by the Generic Solver API, the solver wrapper can extend the base set with *solver-specific actions* which directly interact with the solver API.

Solver Manager. The Solver Manager is the central manager for sorts, operators, and terms created by the solver. It exposes an interface for actions to (i) randomly pick enabled sorts and operators based on certain criteria, and (ii) notify the manager of new terms and sorts. The Solver Manager further maintains *solver-specific configurations* of supported theories, sorts, and operators. It configures solver-specific behavior by querying the solver wrapper to obtain solver-specific configuration information (e.g., solver-specific operators) and restrictions (e.g., unsupported sorts and operators).

Generic Solver API. The Generic Solver API provides a common solver interface for interacting with a solver under test. It covers the majority of the features defined in SMT-LIB, and defines abstract base classes for sort, term and solver implementations. It further provides an interface for configuring the Solver Manager as mentioned above. Integrating a new SMT solver into Murxla amounts to implementing these three classes, and optionally, solver-specific configurations, in a solver wrapper.

The Generic Solver API aims at being as general as possible while supporting all semantic features of the SMT-LIB data model. The Generic Solver API further supports "meta" solvers for different purposes. Murxla implements meta solvers for: (i) performing checks of witnesses that require additional solver instances (model value, unsat core, and unsat assumptions checks); (ii) checking the results of one solver instance against another to identify soundness issues; (iii) translating API call sequences to the SMT-LIBv2 format; and (iv) SMT-LIBv2 input fuzzing of SMT solver binaries in interactive SMT-LIB mode.

3.2 API Fuzzer

The *API Fuzzer* is responsible for generating random but valid API call sequences and is the central component of Murxla. Valid API call sequences are generated based on an API model which is implemented as a weighted finite-state machine (FSM), where states correspond to the current state of the SMT solver, and transitions have a weight, a pre-condition, and an associated action. Each state of the FSM may provide a pre-condition that defines when it is legal

to transition into that state. Taking a transition also executes its action. The associated action of a transition may be empty, in which case it leads to the next state without calling the solver. The pre-condition of a transition and the pre-condition of its next state define the conditions under which the transition can be selected, whereas its weight determines the probability of it being taken in cases where multiple transitions are enabled at the same time.

By default, the FSM implements an API model that captures the functionality and constraints defined in the SMT-LIB standard. And as described above, its associated actions call the Generic Solver API. Murxla supports arbitrary solver-specific modifications to this FSM by providing a configuration interface for solver wrappers (which we discuss in Sect. 3.3 below).

Configuration of the API Fuzzer and execution of its FSM to generate API call sequences for a single run is performed using the following steps.

1. The solver wrapper makes solver-specific modifications to the FSM.
2. The API Fuzzer picks a set of enabled theories, with or without quantifiers.
3. The Solver Manager queries the solver wrapper via the Generic Solver API to configure solver-specific extensions and restrictions.
4. The FSM and Murxla's core components are finalized, and the FSM is set to its initial state; this also creates and initializes the actual solver instance.
5. Next, a set of compatible solver options is selected and configured.
6. After that, the API fuzzer chooses an execution of the FSM and executes the actions associated with that execution, thereby generating a sequence of calls to the solver. This continues until either the solver crashes, the final state is reached, or a configured time limit is exceeded.

In contrast to some recent mutation-based SMT-LIBv2 input fuzzing approaches [39,49,51,52], the API Fuzzer is *generation-based*: it generates expressions that, importantly, respect the semantic and API models of the solver under test. Non-leaf terms are generated by combining leaf terms (variables or theory-specific constants) and previously generated terms via any of the enabled operators. To bias the generated terms towards more variety and structure, each term maintains a reference count, and terms with lower reference counts are selected with a higher probability when constructing new terms. For indexed operator kinds (e.g., the `extract` operator in the theory of fixed-size bit-vectors), random integer values up to a configured maximum value (if not otherwise restricted by the semantics of the operator) are selected. Similarly, arguments to sort constructors (e.g., `Array`) are sampled from previously generated sorts, and sorts with numeric parameters (e.g., bit-vector and floating-point sorts) are constructed from randomly selected integer values up to a configured maximum value.

The API fuzzer utilizes a random number generator (RNG) for random decisions, which is deterministic in the sense that it is guaranteed to produce the same sequence of values when given the same starting seed. The API fuzzer supports two usage modes: (i) single run, starting with a specific seed; and (ii) continuous, consisting of repeated single runs with seeds selected by a dedicated Seed Generator, which uses the current time and process ID to generate seeds.

Each mode can be restricted to a given set of theories (with or without quantifiers) via the command line (in this case, step two of the fuzzer configuration detailed above is skipped). When in *single run mode*, Murxla by default sends a trace of the run to stdout (and optionally to a file). In *continuous mode*, each run is first executed without tracing. If a run uncovers an issue, it is replayed with the same seed and recorded to a trace file. In this mode, Murxla maintains a statistics summary with the current number of issues, timeouts, and *sat*, *unsat*, and *unknown* results. When an issue is discovered, it reports the corresponding seed and solver output. On termination, it provides an overview of all issues, *deduplicated* based on fuzzy matching on the solver output.

Murxla only reports false positives in rare cases where false positives may only be avoided with unreasonable effort, e.g., implementing well-formedness checks for algebraic datatypes.

3.3 Solver Wrappers

As mentioned above, a solver wrapper is used to connect Murxla to a solver. Solver wrappers are typically 2k–4k LOC in size and implement the Generic Solver API. If a solver provides features that are different from those covered by the Generic Solver API, a solver wrapper can accommodate these differences by reconfiguring the FSM of the API Fuzzer to add or remove states, transitions, and actions (added actions can be configured to call the API of the solver under test directly). Solver wrappers are further responsible for configuring the semantic model of the API Fuzzer by (i) adding or removing supported theories and their corresponding sorts and operators; and (ii) extending or restricting the set of operators for supported theories. Solver wrappers may also implement sanity checks of arbitrary complexity by utilizing the `MURXLA_TEST` macro.

The option model of a solver is implemented as part of the Generic Solver API. For Bitwuzla, Boolector, and cvc5, this amounts to 15–55 LOC since all three can be queried for available options and valid configuration values via the API. This allows an automated registration of options with the Solver Manager. Yices2 does not provide this feature which requires that options are registered explicitly. Note that its option model is currently not implemented.

Each solver wrapper maintains its own RNG which is used to make choices when there are multiple alternative solver API calls for one specific task. This RNG is independent from the main RNG of the API Fuzzer and is seeded with a value generated by the main RNG for each action execution. These seeds are recorded by the Tracer to ensure that random choices can be deterministically replicated when replaying a traced run of the API Fuzzer.

3.4 Tracer, Untracer, Trace Minimizer

The **Tracer** records all action executions with their corresponding arguments and return values in a text-based format. Each action line in the trace follows the pattern `<seed> <action> [<args...>]`, optionally followed by a return statement of the form `return <values...>` for actions that create sorts or terms. The `<seed>`

```
74761 new
65471 set-logic QF_BV
33949 mk-sort SORT_BOOL
      return s1
64345 mk-sort SORT_BV 8
      return s2
49391 mk-const s2 "a"
      return t1
89712 mk-const s2 "b"
      return t2
 6548 mk-term OP_EQUAL SORT_BOOL 2 t1 t2
      return t3 s1
20351 assert-formula t3
47017 check-sat
74496 delete
```

Fig. 2. Murxla trace for checking $a = b$ for bit-vectors a and b of size 8.

in an action is the seed of the solver wrapper's RNG when executing the action. It is recorded to ensure that random choices made by the solver wrapper can be deterministically replicated. This is especially important when minimizing a trace, since modifying trace lines may change the way the main RNG behaves when replaying the trace. Sorts are recorded as `s<id>` and terms as `t<id>`, and the `<args...>` of an action line determine all sort, term and numerical arguments required to replay the execution of the given action. Similarly, the `<values..>` of a return statement record all of its sort and term return values. Figure 2 shows an example of a trace generated by Murxla. It records the action sequence for checking the satisfiability of $a = b$, where a and b are bit-vectors of size 8. Note that when creating terms via `mk-term`, we trace argument lists while also providing the number of arguments, e.g., `2 t1 t2`. The same applies for indices of indexed operators. Further, for any action that creates terms that are added to the term database (e.g., `mk-term`), we also need to trace the sort of the created term, e.g., `return t3 s1`. This is due to the fact that some operators create terms of new sorts that may not have been encountered in the trace yet.

The **Untracer** takes a trace as input and replays each recorded action, thereby replicating the behavior of the original execution. This is especially useful for debugging erroneous behavior of the solver under test. Additionally, if a trace does not contain any solver-specific extensions, the Untracer can replay it using a different solver or translate it to the SMT-LIBv2 format. Tracing actions instead of calls to the Generic Solver API has the advantage that both the API Fuzzer and the Untracer can use the same infrastructure for communicating with the solver under test. Furthermore, supporting solver-specific actions does not require changes to any component other than the solver wrapper.

The **Trace Minimizer** is built on top of the Untracer and minimizes a given trace while preserving the behavior of the original execution. It implements simple ddmin-style [53] minimization techniques in three phases: (i) line-based

minimization to reduce the number of trace lines; (ii) minimization of action lines to reduce the number of arguments; and (iii) term substitution, where terms are replaced with simpler terms of the same sort. Even though all of these minimization techniques are rather basic, the Trace Minimizer typically reduces the size of a trace to less than 10% of the original trace. If the minimized trace can be translated to SMT-LIB, then it can often be further reduced using a delta-debugging tool such as ddSMT [37]. Even if a minimized trace cannot be expressed in SMT-LIB due to solver-specific extensions, we have found that in practice, the reduction due to the Trace Minimizer is typically good enough to allow efficient debugging.

4 Evaluation

We evaluate the efficacy of Murxla in three experiments, comparing: (1) Murxla and BtorMBT, testing Boolector; (2) Murxla and the current state-of-the-art input fuzzers STORM [39] and TypeFuzz [47]; and (3) Murxla with and without option fuzzing. For this evaluation, we target soundness issues and crashes, and do not consider performance regressions. In the following, we use *issues* to mean crashes unless explicitly otherwise noted. We use Bitwuzla commit eea0973 [5], Boolector commit b157b10 [6], cvc5 commit 0f5ee6b [7], and Yices2 commit 09f1621 [8]. For each experiment we compare the number of issues uncovered by each tool, and the code coverage of the solver under test. Code coverage was measured with gcov, which is part of the GNU Compiler Collection [9]. We performed all experiments in an Ubuntu 21.04 Docker container on a machine with an AMD Threadripper 3970X CPU and 128GB of memory and used a one hour wall-clock time limit for each experiment and tool.

Murxla vs. BtorMBT. We compare the effectiveness of fuzzing Boolector with Murxla against that of its own custom API fuzzer BtorMBT. We ran both tools in continuous mode with a one second time limit per single run. Murxla achieves a line (function) coverage of 81% (88%) and finds 18 issues (including 3 known reported issues). BtorMBT achieves 72% (81%) coverage, but does not find any issues. BtorMBT does not support quantifiers, and three of the issues found by Murxla are located in Boolector's quantifiers module. The other issues, however, occur in code that is covered by BtorMBT.

Murxla vs. STORM, TypeFuzz. We test cvc5 with Murxla, STORM, and TypeFuzz on QF_SLIA problems. We use all QF_SLIA benchmarks in the SMT-LIB benchmark library as seed files for STORM and TypeFuzz. Both Storm and TypeFuzz mainly target soundness issues. TypeFuzz requires using at least two SMT solvers as it relies on comparing their results, whereas Storm creates satisfiable formulas by construction and does not require cross-checking. Hence, we additionally use a cross-checking configuration of Murxla (*Murxla-cc*), which compares Z3 version 4.8.14 and cvc5. Since Murxla does not yet integrate Z3, we use it via Murxla's SMT-LIBv2 interface in interactive SMT-LIB mode (the input fuzzing mode). Note that this requires disabling solver-specific extensions

Table 1. Number of issues (I), and line (L) and function (F) coverage for experiments two (top) and three (bottom). Option fuzzing for Yices2 is not yet implemented (-).

Murxla			**STORM**			**Murxla-cc**			**TypeFuzz**		
L [%]	F [%]	I	L [%]	F [%]	I	L [%]	F [%]	I	L [%]	F [%]	I
37.8	52.5	7	20.2	34.3	0	21.5	36.3	1	17.4	30.8	0

Option	**Bitwuzla**			**Boolector**			**cvc5**			**Yices2**		
Fuzzing	L [%]	F [%]	I	L [%]	F [%]	I	L [%]	F [%]	I	L [%]	F [%]	I
no	47.4	63.9	7	68.5	79.2	6	38.9	56.8	11	37.0	42.4	1
yes	62.9	75.8	23	81.1	87.7	13	49.1	66.8	21	-	-	

of cvc5, since they are unsupported by Z3. The results are shown in Table 1. Murxla and Murxla-cc have consistently higher coverage than the other tools and find 8 issues, whereas the other tools find none. Most notably Murxla-cc was able to find a model unsoundness issue in cvc5, where cvc5 incorrectly reports *satisfiable* due to an incorrect rewrite rule for the `re.loop` operator [1].

Option Fuzzing. We evaluate the effectiveness of Murxla with and without option fuzzing on all supported solvers. We use the default configuration of Murxla, which tests all supported features for each solver. The results are shown in Table 1 and showcase the efficacy of option fuzzing both for improving coverage and for finding issues. In its best configuration, Murxla achieves an API function coverage of 85% for Bitwuzla, 94% for Boolector, 68% for cvc5, and 46% for Yices2. cvc5 provides the richest API, supporting not only all of SMT-LIB but also non-standard theories and non-SMT features like SyGuS and high-order reasoning, which are not yet supported in Murxla. Bitwuzla and Boolector export parsing via the API, which is currently only supported in Murxla for Boolector. Coverage for Yices2 is low in comparison as it was integrated as a proof of concept, and its wrapper does not yet implement all of its features nor its option model.

The artifact containing the experimental data of this evaluation is available at https://zenodo.org/record/6494381.

5 Conclusion

Our experimental evaluation shows that Murxla quickly and effectively finds issues in multiple state-of-the-art SMT solvers—even for logics like QF_SLIA which have been the subject of month-long fuzzing campaigns [39,47,51,52] over the last two years. Furthermore, during the past few months, while finalizing and testing Murxla, we found many more issues in these solvers—more than 100 for cvc5 alone, and some of them critical [3,4]. Based on this success, we believe that Murxla will be a valuable tool for stress-testing SMT solvers and thereby improving their correctness and robustness. We are currently in the process of integrating it into the development workflow of Bitwuzla, Boolector and cvc5.

References

1. cvc5 model unsoundness issue found by Murxla-cc. https://github.com/cvc5/cvc5-projects/issues/409
2. Boolector issue tracker (2022). https://github.com/boolector/boolector/issues
3. cvc5 issues found by Murxla, reported on internal issue tracker (2022). https://github.com/cvc5/cvc5-projects/issues?q=is:issue+is:open+label:murxla
4. cvc5 issues found by Murxla, reported on official issue tracker (2022). https://github.com/cvc5/cvc5/issues?q=is:open+is:issue+label:murxla
5. Bitwuzla GitHub repository (2022). https://github.com/bitwuzla/bitwuzla
6. Boolector GitHub repository (2022). https://github.com/boolector/boolector
7. cvc5 GitHub repository (2022). https://github.com/cvc5/cvc5
8. Yices2 GitHub repository (2022). https://github.com/SRI-CSL/yices2
9. GNU Compiler Collection (2022). https://gcc.gnu.org/
10. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–8. IEEE (2013)
11. Artho, C., Biere, A., Seidl, M.: Model-based testing for verification back-ends. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 39–55. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_3
12. Backes, J., et al.: Stratified Abstraction of Access Control Policies. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 165–176. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_9
13. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS (1). LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
14. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
15. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
16. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
17. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the sat competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) SAT Competition 2017 - Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)
18. Bjørner, N.: SMT in verification, modeling, and testing at microsoft. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 3–3. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_3
19. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: a fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 45–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_6
20. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient smt-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 151–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12

21. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: SMT, pp. 1–5 (2009)
22. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. USENIX Association (2008)
23. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
24. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
25. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating smt solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
26. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
27. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
28. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: `SMT-RAT`: an open source `C++` toolbox for strategic and parallel smt solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
29. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
30. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
31. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
32. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
33. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
34. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI, pp. 62–73. ACM (2011)
35. Hajdu, Á., Jovanović, D.: solc-verify: a modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 161–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_11
36. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: an smt solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_35

37. Kremer, G., Niemetz, A., Preiner, M.: ddSMT 2.0: better delta debugging for the smt-libv2 language and friends. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 231–242. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_11
38. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
39. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: ESEC/SIGSOFT FSE, pp. 701–712. ACM (2020)
40. Mattarei, C., Mann, M., Barrett, C.W., Daly, R.G., Huff, D., Hanrahan, P.: Cosa: Integrated verification for agile hardware design. In: FMCAD, pp. 1–5. IEEE (2018)
41. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
42. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020 (2020). CoRR abs/2006.01621
43. Niemetz, A., Preiner, M.: Murxla (2022). https://github.com/murxla/murxla
44. Niemetz, A., Preiner, M.: Murxla Documentation (2022). https://murxla.github.io
45. Niemetz, A., Preiner, M., Biere, A.: Model-based API testing for SMT solvers. In: SMT. CEUR Workshop Proceedings, vol. 1889, pp. 3–14. CEUR-WS.org (2017)
46. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 587–595. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_32
47. Park, J., Winterer, D., Zhang, C., Su, Z.: Generative type-aware mutation for testing SMT solvers. In: Proc. ACM Program. Lang. (OOPSLA), vol. 5, pp. 1–19 (2021)
48. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Tech. rep., Department of Computer Science, The University of Iowa (2006)
49. Scott, J., Sudula, T., Rehman, H., Mora, F., Ganesh, V.: BanditFuzz: fuzzing SMT solvers with multi-agent reinforcement learning. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 103–121. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_6
50. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
51. Winterer, D., Zhang, C., Su, Z.: On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. Proc. ACM Program. Lang. (OOPSLA), vol. 1, pp. 193:1–193:25 (2020)
52. Winterer, D., Zhang, C., Su, Z.: Validating SMT solvers via semantic fusion. In: PLDI, pp. 718–730. ACM (2020)
53. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. **28**(2), 183–200 (2002)