Fuzzing and Delta-Debugging SMT Solvers

Robert Daniel Brummayer

Fuzzing and Delta-Debugging SMT Solvers

Robert Brummayer and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Abstract. SMT solvers are widely used as core engines in many applications. Therefore, robustness and correctness are essential criteria. Current testing techniques used by developers of SMT solvers do not satisfy the high demand for correct and robust solvers, as our testing experiments show. To improve this situation, we propose to complement traditional testing techniques with grammar-based blackbox fuzz testing, combined with delta-debugging. We demonstrate the effectiveness of our approach and report on critical bugs and incorrect results which we found in current state-of-the-art SMT solvers for bit-vectors and arrays.

1 Introduction

Many applications use Satisfiability Modulo Theories (SMT) solvers as core decision engines. For example, SMT solvers are used to generate test cases, to find bugs [5,11,12,30,31], and to verify systems [2,6,19,20,21,23]. A crashing SMT solver may lead to a crash of the application, or even worse, an incorrect solver may lead to wrong results. For example, if an SMT solver concludes *unsat* although the input formula is *sat*, a verification system may spuriously conclude that an implementation respects its specification, i.e. defects are missed.

In this paper we show that although there is a high demand for robustness and correctness of SMT solvers, almost all state-of-the-art solvers, at least for bit-vectors and arrays, were broken at the time of our tests. They contained defects that led to crashes, or even worse, to incorrect results where the solver concludes *sat* although the formula is *unsat*, or vice versa. We demonstrate that many critical defects can be found by a testing technique called *grammar-based blackbox fuzz testing*, and propose to complement traditional testing approaches with this technique. Moreover, we propose to integrate *delta-debugging* [34] into the debugging process in order to minimize failure-inducing SMT formulas.

2 Fuzzing

Fuzzing is a powerful testing technique which is typically used in the domains of software security and quality assurance [28,29]. The main idea of the original fuzzing approach is to test programs with random inputs in order to detect security bugs, e.g. buffer overflows. Fuzz testing techniques were already applied by software engineers around 1980. For example, a tool called "the monkey" was developed to test the original Macintosh system. It fed random events to the current application. It appeared as if the computer was operated by an angry monkey [29]. In [24], fuzz testing was used to find bugs in UNIX tools.

The first tools were simple, used *blackbox* testing, i.e. testing was performed against the interface without access to implementation details, and had no knowledge of the expected input format. In the context of SMT solvers we are typically not interested in fuzzing techniques that are unaware of the input syntax. Although useful, such techniques would mainly test syntax error handling routines of the parser[1]. We would only scratch the surface and would not be able to test deeper parts of the solver. However, in this paper we focus on finding critical bugs such as crashes upon syntactically valid inputs, and incorrect results.

Nowadays, various fuzzing tools are available [28,29]. Moreover, there is an ongoing research on *whitebox* fuzz testing tools, which are a new generation of sophisticated fuzzers that use recent advances in symbolic execution and dynamic test generation [17,18]. Whitebox fuzzing is an interesting application for SMT solvers. However, in this paper we use fuzzing techniques in order to *test* solvers. Although the *whitebox* fuzzing approach seems promising for testing solvers, it has its limitations when it is used for programs that expect highly structured inputs such as the SMT format [26]. Sophisticated and complicated techniques like *grammar-based whitebox fuzzing* [17] are necessary in order to use *whitebox* techniques for testing deep parts of SMT solvers, e.g. error prone optimizations.

We propose to use *grammar-based blackbox fuzzing* for testing SMT solvers. A fuzzer randomly generates syntactically valid SMT formulas in order to detect critical defects. Unlike *grammar-based whitebox fuzzing*, *grammar-based blackbox fuzzing* is easy to implement and integrate, generates no false positives, and is impressingly effective in finding bugs that traditional testing techniques miss. This is confirmed by our experiments in section 4. Moreover, in contrast to fuzzing techniques such as [13], it does not need any user specifications and can be fully automated.

2.1 Generating Random Bit-Vector Formulas

We use a layered approach for generating random formulas, similar to [32]. In the following we focus on bit-vector formulas. However, the main concepts and ideas can also be used in the context of other theories.

Although the main algorithm is rather simple, many details have to be considered as the quantifier-free theory of fixed-size bit-vectors has many different operators. While some of them use bit-vector arguments only, other operators, e.g. `extract`, also use integer constants. Moreover, the operators require different preconditions, e.g. equal bit-width of the operands, valid index positions, etc., which have to be fulfilled. In contrast to [32], we also have to consider more types, i.e. unlike the

[1] Try to pipe `cat /dev/urandom` to an arbitrary SMT solver.

BTOR format [8], the SMT-LIB format [26] distinguishes between type *boolean* and *bit-vector of bit-width one.*

In principle, we structure a bit-vector formula into four layers: *input*, *main*, *predicate*, and *boolean.* During the formula construction we maintain a set of all nodes that have been created, including their types.

First, we generate the *input* layer with a random number of bit-vector variables and constants. The bit-width is also selected randomly which makes generating random bit-vector formulas more complicated than formulas that contain natural numbers. We have to consider many different types as we do not want to restrict all bit-vector terms to the same bit-width.

Second, we generate the *main* layer. We iteratively choose either a random bit-vector operator[2], or one of {`=`, `distinct`, `ite`}. Then, depending on the arity of the operator, we randomly select operands from our set, generate the final node, and insert it into the set. The main problem in this step is that the operands might have different bit-widths, but almost every bit-vector operator requires them to be equal. We use the extension operators `zero_extend` and `sign_extend`, and the extraction operator `extract` to solve this problem. If the operands do not have the same bit-width, then we either extend the "smaller" operand, or select a sub-vector of the "larger" operand.

In the *main* layer we are interested in bit-vector nodes only. However, some operators, i.e. `=`, `distinct`, and bit-vector predicates, result in boolean nodes. In order to convert them to bit-vector terms we use an if-then-else wrapper. Assume t_1 and t_2 are bit-vector terms with the same bit-width, and p is a bit-vector predicate. We convert $p(t_1, t_2)$ into a bit-vector term as follows:

$$(ite\ (p\ t_1\ t_2)\ bv1[1]\ bv0[1])$$

If $p(t_1, t_2)$ is true, we return the bit-vector constant one with bit-width one, and zero otherwise. Hence, we can use wrapped predicates as inputs to bit-vector operations. This technique aims to detect subtle bugs that are not found by tests that use predicates in a boolean context only.

Third, we analogously generate the *predicate* layer. However, we restrict our operator selection to `=`, `distinct`, and bit-vector predicates. Finally, we generate the *boolean* layer by randomly combining boolean nodes to one final root. We iteratively select roots, i.e. boolean nodes without parents, from our boolean layer, and combine them by a random boolean operator. We continue this process until there is only one root left.

2.2 Bit-vector Arrays

Adding one-dimensional bit-vector arrays is straightforward. We extend our algorithm for generating random bit-vector formulas as follows. First, we add array

[2] In the SMT-LIB there are 35 bit-vector operators in `QF_BV`.

variables to the *input* layer. Then, during the process of building the *main* layer, we also build an *array* layer by using $\mathit{write}(a, i, e)$, where a is an array, i a bit-vector index and e a bit-vector value. The result of $\mathit{write}(a, i, e)$ is an *array* where the value at position i has been overwritten by e. All other elements of a remain the same. We select a, i, and e randomly from the terms that have already been created. If the bit-widths of i and e are incompatible to a, we use the techniques described earlier.

Moreover, while we are building the *main* bit-vector and array layer we also create reads by using $\mathit{read}(a, i)$, where a is an array and i is a bit-vector index. Analogously to $\mathit{write}(a, i, e)$, we select a and i randomly from the terms that have been created before, and either extend or slice i if necessary.

Interleaving the phases of creating regular bit-vector terms, reads and writes ensures that reads are also used as read indices, write indices, write elements, and also as operands of regular bit-vector operations. Moreover, nested writes may be created. Generating such formula structures aims to find subtle bugs in array algorithms that use abstraction refinement loops [7,15] where reads are internally replaced by fresh variables.

If we want to support the extensional theory of arrays where we can compare arrays in addition to array elements, then we can extend the *main* bit-vector layer with array equalities, encoded as bit-vectors. Moreover, array equalities may also be added to the *boolean* layer.

3 Delta-Debugging SMT Formulas

After we have found failure-inducing inputs, we typically want to minimize them. Delta-debugging techniques [1,13,25,34,35] *automatically* simplify and isolate failure-inducing inputs by using a divide-and-conquer strategy. Typically, minimized inputs speed up debugging as non-irrelevant input parts do not have to be considered. Moreover, large inputs may be practically infeasible to debug.

A delta-debugger repeatedly calls the program with simplified variants of the failure inducing input. If the program shows the same observable behavior, e.g. returns the same exit code or prints out the same error message, the delta-debugger continues with the simplified input, and backtracks otherwise.

Generally, delta-debugging does not generate a minimal failure-inducing input. However, this feature is rarely needed in practice. Typically, the goal of delta-debugging is to reduce the input as much and as fast as possible. It is not feasible for engineers to wait for a delta-debugging tool that needs days or even weeks to terminate. Therefore, we use a small and simple set of simplifications which allows fast delta-debugging while generating very small failure-inducing inputs. This is also confirmed by our experiments in Tab. 3.

In the context of SMT we have highly structured inputs, and type information. As Zeller's original delta-debugging technique [34] does not explicitly use any knowledge of the input structure, we use a variant of hierarchical delta-debugging [25].

Hierarchical delta-debugging techniques have also been proposed for BTOR [32]. We use the knowledge of formula structures and types to speed up the delta-debugging process, and to minimize inputs even further.

First of all, we represent an SMT formula as DAG with one boolean root. Typically, SMT formulas are layered, e.g. there is a boolean layer on top of the formula. We use the knowledge of a boolean layer to prune large irrelevant parts of the input up front. We iteratively try to replace the current root by one of its boolean children, i.e. we perform a search through the boolean layer.

Then, we perform further term-level simplifications, driven by a breath-first-search. Whenever new nodes are created, we insert them into a queue for further simplifications. We try to substitute each node either by the constant zero, one, or by one of its children, but only if the types of the current node and its child match. Note that this is not possible for bit-vector predicates, as they have a boolean type while their children have bit-vector types. If one child is a chain of unary operator applications, or writes, we try to skip it, i.e. we try to replace the current node by the node at the end of the chain. By using this technique we may immediately prune irrelevant operator chains, e.g. deeply nested writes.

Finally, after all nodes have been processed, we try to find a new root again. Boolean nodes may not only occur within the *boolean* layer, but also deeper inside the formula, used as conditions in if-then-else operators. We expect that the input formula has now been simplified significantly by the delta-debugging process. Therefore, we can try to substitute the current root by arbitrary boolean nodes that occur deeper in the formula. In this way we may minimize the formula even further and eliminate irrelevant if-then-else nodes in upper formula layers.

3.1 Delta-Debugging Crashes

Typically, fuzz testing leads to a high number of system crashes, i.e. the program terminates without providing a result. This is also confirmed by our experiments in section 4. The randomness of the input is responsible for triggering statements that have not been tested before. Executing untested statements may lead to erroneous internal states, and thus, crashes.

In the context of SMT solvers, we have observed that crashes typically occur almost immediately after starting a solver. We conjecture that this observation also holds for other kinds of solvers that use complex and error prone optimization techniques, e.g. SAT solvers. We can use this observation to improve the performance of delta-debugging.

First, we introduce the concept of timeouts. During delta-debugging, each call to the solver is executed using a time limit. Whenever the solver exceeds its limit, we treat this case as if the simplification of the failure-inducing input has failed, and backtrack. Note that using timeouts during delta-debugging can also be useful for other kind of defects. For example, it can be used for delta-debugging failure-inducing formulas that lead to an infinite loop within the solver.

Whenever we want to delta-debug a formula that leads to a crash, we typically set the time limit to a few seconds above the time which leads to a crash on the original formula. Using timeouts may heavily speed up delta-debugging formulas that are hard to decide. For example, assume we want to delta-debug a complex SMT formula. Whenever a specific sub-formula structure occurs, the solver crashes almost immediately. However, whenever delta-debugging simplifications destroy this sub-formula structure, the solver works correctly and may run for days. By using timeouts we can backtrack simplifications that do not lead to a crash almost immediately, and thus, speed up delta-debugging significantly, i.e. the delta-debugger may terminate with a small failure-inducing input already within a few minutes.

4 Experiments

To evaluate the effectiveness of our approach, we fuzz-tested and delta-debugged publicly available state-of-the-art SMT solvers with our fuzzer `FuzzSMTBV` and our delta-debugger `DeltaSMT`. The failure-inducing inputs and delta-debugged results are available at `www.fmv.jku.at/brummayer/fuzz-dd-smt.tar.7z`.

We ran our experiments under Ubuntu Linux on an Intel Core 2 Quad machine with 2.66 GHz and 8 GB RAM. Our fuzzing test framework used each of the four cores for testing. Our delta-debugging experiments were performed on the same machine, but were not run simultaneously.

The process of testing SMT solvers was rather complicated and complex. At the time of our tests, only two solvers, Boolector [7] and Z3 [14], supported all bit-vector operators of the SMT-LIB [27] without crashing. Therefore, we used Boolector as a filter to rewrite high-level operators, e.g. smod, into a combination of supported low-level base operators according to [8]. Although we had to restrict the tests to at most 11 out of 35 bit-vector operators, we found an impressive number of bugs. We conjecture that we would have found even more bugs if we had been able to use the full set of operators.

For the quantifier-free theory of bit-vectors `QF_BV` we tested the following solvers: Beaver [22] 1.1-RC1, Boolector [7] 1.0 and 1.1, a development version of CVC3 [4] 1.5 (downloaded April 29th, 2009), MathSAT [9] 4.2.3, Spear [2] 2.7 with SMT2SF 1-9, Sword [33] from SMT-COMP'08 [3], Z3 [14] 1.2, Z3 from SMT-COMP'08 [3], and an unstable internal version of OpenSMT [10]. The results are summarized in Tab. 1. Note that the current versions of CVC3 and OpenSMT do not support bit-vector division. Moreover, we could not test STP for bit-vector formulas as we encountered serious problems when we tried to use CVC3 to convert SMT formulas to CVC format which is needed by STP. However, we could test STP for a more restricted bit-vector logic combined with arrays. These results are shown in Tab. 2.

We detected incorrect results, i.e. solvers report *sat* although the status is *unsat* or vice versa, in the following way. First, we compared the result of each solver on each formula to the result of Boolector 0.4 and Z3 from SMT-COMP'08. If Boolector

	no-div		guard-div	
solver	crash	incorrect	crash	incorrect
Beaver 1.1 rc1	0	0	12430	1
Boolector 1.0	0	0	0	0
Boolector 1.1	0	0	0	0
CVC3 1.5	902	8	-	-
MathSAT 4.2.3	0	113	2097	83
OpenSMT	19871	8	-	-
Spear 2.7	0	6	3577	71
Sword smt-comp	0	1	0	0
Z3 1.2	0	0	2264	0
Z3 smt-comp	0	0	0	0

Table 1. Experimental results of fuzzing bit-vector solvers. The file size of random SMT formulas typically ranges from a few KB to 1 MB. The results are divided into bit-vector formulas without division operators (`no-div`) and formulas with "guarded" division (`guard-div`). Moreover, the results show the number of crashes, i.e. solver terminates in an unexpected way without providing a result, and number of incorrect results, i.e. solver reports *unsat* although formula is *sat*, or vice versa. Guarded division adds top-level constraints that rule out models where division by zero occurs. This guarantees that the semantics of dividing by zero does not influence the satisfiability status of the formula. We used a maximum bit-width of 16 for `no-div` formulas and tested each solver with the same set of 23100 randomly generated SMT formulas in three hours. For `guard-div` formulas that may additionally contain `bvudiv` and `bvurem`, we used a maximum bit-width of 10, as bit-vector division significantly slowed down some solvers. In this category we tested 23100 formulas in about one hour.

and Z3 agreed on the result[3], but the tested solver reported the opposite, we collected this formula.

Then, we inspected the collected formulas. Beaver claims that one formula is *sat* although other solvers report *unsat*. However, Beaver crashes with an assertion failure when asked to provide a model. For CVC3 we could use its built-in on-the-fly proof checker to confirm nearly all cases where CVC3 wrongly concludes *unsat*. Moreover, most of the remaining wrong answers are caused by CVC3's query preprocessor. After disabling preprocessing, CVC3 reports the expected answer in most of the cases. For MathSAT we used the command line argument `-smtcomp` as it is documented in the MathSAT 4 invocation guide on their webpage. However, we found out that this argument is responsible for many incorrect results where MathSAT spuriously reports *unsat*. If we use `-input=smt`, `tsolver=bv` and `-solve` instead of `-smtcomp`, MathSAT reports the expected results in almost every case. Similarly, Spear reports the expected results when common subexpression elimination is turned off via `--cse 0`. The remaining incorrect results were confirmed by the majority voting principle where we used at least Boolector 1.1, Z3 from SMT-COMP'08 and another solver where error prone optimizations were turned off.

[3] We could not find any formulas where Boolector and Z3 disagreed.

For the quantifier-free theory of bit-vectors, arrays and uninterpreted functions `QF_AUFBV`, we tested the following solvers: Boolector [7] 1.0 and 1.1, a development version of CVC3 [4] 1.5 (downloaded April 29th, 2009), STP [16] 0.1 (November 18th, 2008), and Z3 [14] 1.2 and from SMT-COMP'08 [3]. The results are summarized in Tab. 2.

solver	crash
CVC3 1.5	9812
STP 0.1	24

Table 2. Experimental results of fuzzing bit-vector and array solvers. The formulas contain bit-vector arrays, reads and writes, but no equalities between arrays as STP does not support them. We tested each solver with the same set of 12000 randomly generated formulas in about two hours. In order to be able to use CVC3 as filter to test STP, we had to restrict the set of operators which was used for the results in Tab. 1 even further, i.e. the formulas neither contain division nor shift operators. For Boolector and Z3 we could not find any defects. Generally, no incorrect results were found, but serious internal crashes. An analysis of the error messages showed that some crashes were caused by defects in the implementation of decision procedures.

Finally, we delta-debugged failure-inducing formulas found for `QF_BV`. Before delta-debugging, we semi-automatically divided them into bug classes with a limit of 50 formulas for each class. The results are shown in Tab. 3. We encountered only a few non-deterministic bugs that we were not able to delta-debug as they were not always reproducible. Incorrect results were delta-debugged as follows. Instead of calling the incorrect solver directly, the delta-debugger calls a shell script during delta-debugging. The script calls three trusted solvers and the incorrect solver. It returns 1 only if the three trusted solvers agree on the satisfiability status and the incorrect solver reports the opposite, and 0 otherwise.

5 Conclusion

The goal of this paper is to show that traditional testing techniques obviously do not suffice to fulfil the high demand for robustness and correctness of SMT solvers. SMT solvers contain a lot of error prone optimizations that need to be heavily tested. Although fuzz testing is not a complete solution for this problem, i.e. it can only find bugs but cannot prove the absence, we have shown that it is a useful "tool in the toolbox“ which can find many bugs in state-of-the-art SMT solvers that traditional testing techniques obviously miss.

We conclude that fuzz testing in combination with delta-debugging is an effective approach which can be easily integrated in the development process of SMT solvers in order to increase robustness and correctness. Moreover, we believe that this combination can be of great value in other domains as well. Blackbox fuzzing is able to find many defects, even without any knowledge of implementation details.

	no-div					guard-div				
solver	f	c	t	s	r	f	c	t	s	r
Beaver 1.1 rc1	-	-	-	-	-	469	12	5	319	98%
CVC3 1.5	139	9	172	2429	98%	-	-	-	-	-
MathSAT 4.2.3	50	1	10	611	97%	190	5	58	3709	76%
OpenSMT	154	4	5	492	96%	-	-	-	-	-
Spear 2.7	6	1	5	401	96%	100	2	4	228	99%
Sword smt-comp	1	1	4	135	99%	-	-	-	-	-
Z3 1.2	-	-	-	-	-	50	1	734	254	99%

Table 3. Experimental results of delta-debugging bit-vector solvers. The columns labelled f and c represent the number of formulas, and the number of bug classes. The number of classes is a rough approximation for the number of different solver defects. The columns t, s, and r show the average delta-debugging time in seconds, the average file size in bytes after delta-debugging, and the average file size reduction. We encountered some statistical outliers. The median of time t is 2 seconds for OpenSMT, 14 seconds for CVC3 and 648 seconds for Z3. The medians for time t, file size s after delta-debugging, and reduction in file size r are 13 seconds, 715 bytes and 91% for the `guard-div` examples of MathSAT. Moreover, the median of s is 918 bytes for CVC3.

It is therefore a reasonable conjecture that using knowledge of these details in a whitebox fuzzing approach may be even more successful in finding more, or more subtle, defects.

5.1 Acknowledgements

We want to thank A. Vida for his predecessor work in his Master's Thesis [32] on fuzzing and delta-debugging for BTOR. Moreover, we want to thank R. Bruttomesso for taking the risk to provide an unstable internal version of OpenSMT. Finally, the first author wants to thank L. de Moura and C. Tinelli for their encouragement to perform these experiments and to write this paper.

References

1. C. Artho. Iterative Delta Debugging. In *Proc. HVC*. Springer, 2008.
2. Domagoj Babić. *Exploiting Structure for Scalable Software Verification*. PhD thesis, University of British Columbia, Vancouver, Canada, 2008.
3. C. Barrett, M. Deters, A. Oliveras, and A. Stump. SMT-Comp, 2008.
4. C. Barrett and C. Tinelli. CVC3. In *Proc. CAV*. Springer, 2007.
5. N. Bjørner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *Proc. TACAS*. Springer, 2009.
6. M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzen, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. In *Proc. PDPAR*, 2005.
7. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. TACAS*. Springer, 2009.
8. R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *Proc. BPR*. ACM, 2008.

9. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In *Proc. CAV*. Springer, 2007.
10. R. Bruttomesso and N. Sharygina. OpenSMT 0.1 System Description, 2008.
11. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*. USENIX Association, 2008.
12. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. SIGSAC*, 2006.
13. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. ICFP*. ACM, 2000.
14. L. de Moura and N. Bjørner. Z3: An Efficient SMT solver. In *Proc. TACAS*. Springer, 2008.
15. V. Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Computer Science Department, Stanford University, 2007.
16. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV*. Springer, 2007.
17. P. Godefroid, A. Kiezun, and M. Levin. Grammar-Based Whitebox Fuzzing. In *Proc. PLDI*. ACM, 2008.
18. P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proc. NDSS*. Internet Society, 2008.
19. T. Henzinger, T. Hottelier, and Laura Kovacs. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proc. LPAR*. Springer, 2008.
20. J. Smans, B. Jacobs, F. Piessens and W. Schulte. An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In *Proc. FASE*. Springer, 2008.
21. P. Jackson, B. Ellis, and K. Sharp. Using SMT solvers to verify high-integrity programs. In *Proc. AFM*. ACM, 2007.
22. S. Jha, R. Limaye, and S. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *Proc. CAV*. Springer, 2009. To appear.
23. S. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *Proc. POPL*. ACM, 2008.
24. B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A.Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CS-TR-1995-1268, University of Wisconsin, Madison, 1995.
25. G. Misherghi and Z. Su. HDD: Hierarchical Delta Debugging. In *Proc. ICSE*. ACM, 2006.
26. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006.
27. S. Ranise and C. Tinelli. The satisfiability modulo theories library, 2009.
28. M. Sutton, A. Greene, and P. Amini. *Fuzzing - Brute Force Vulnerability Discovery*. Pearson Education, 2007.
29. A. Takanen, J. Demott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.
30. N. Tillmann and J. de Halleux. PEX - White Box Test Generation for .NET. In *Proc. TAP*. Springer, 2008.
31. D. Vanoverberghe, N. Tillmann, and F. Piessens. Test Input Generation for Programs with Pointers. In *Proc. TACAS*. Springer, 2009.
32. A. Vida. Random Test Case Generation and Delta Debugging for BitVector Logic with Arrays. Master's thesis, Johannes Kepler University, Linz, Austria, 2008.
33. R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like Prover Using Word Level Information. In *Proc. VLSI*. IEEE, 2007.
34. A. Zeller. *Why Programs Fail. A Guide to Systematic Debugging*. Kaufmann, 2005.
35. A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 2002.