Fuzzing and Delta Debugging And-Inverter Graph Verification Tools*

Daniela Kaufmann[1] ✉, Armin Biere[2]

[1] Johannes Kepler University Linz, Austria, & Software Competence Center Hagenberg, Austria daniela.kaufmann@scch.at
[2] Albert-Ludwigs-University Freiburg, Germany, biere@cs.uni-freiburg.de

Abstract. Ensuring correctness of verification tools is equally important as the correctness of the actual problems they try to establish. In this paper we evaluate automated fuzzing and debugging techniques applied to state-of-the-art multiplier verification tools, which take the common gate-level representation of and-inverter graphs as input. With a generation- and mutation-based fuzzing approach our tools are able to uncover major faults in verification tools including crashes as well as inaccurate verification results. Additionally we demonstrate the usefulness of certificates for automated reasoning tools. We further apply delta debugging techniques and show their effectiveness in reducing failure-inducing inputs.

1 Introduction

Hardware verification and particularly formal verification of arithmetic circuits is important to prevent issues like the infamous Pentium FDIV bug [31]. In recent years verification of gate-level integer multipliers has made significant progress and it has been shown that algebraic reasoning techniques are particularly successful [6, 14, 20, 21]. In this line of work the given multiplier circuit is modeled as a set of polynomials that generates a Gröbner basis. Using a polynomial reduction algorithm, it is checked whether the specification, also modeled as a polynomial, is implied by the circuit polynomials. We refer the reader to [11] for a more detailed introduction to circuit verification using computer algebra.

Sophisticated reduction engines geared towards the automatic verification of multipliers are implemented in the two state-of-the-art tools DyPoSub [21] and AMulet2 [13]. These tools receive multiplier circuits, given as and-inverter graphs (AIGs) [18], internally generate a polynomial encoding and perform the reduction. DyPoSub simply provides a yes-or-no answer on the correctness of the circuit. Our tool AMulet2 additionally provides a proof certificate for correct circuits, and counterexamples in the case of an incorrect circuit.

* This work is supported by the LIT AI Lab funded by the State of Upper Austria, by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the Province of Upper Austria in the frame of the COMET–Competence Centers for Excellent Technologies Programme and the COMET Module DEPS managed by the Austrian Research Promotion Agency FFG.

Proofs and concrete counterexamples (models) provide an additional layer of confidence for the yes-or-no answer of the verification tool. Especially, as we will see in our experimental evaluation, DyPoSub and AMulet2 do not always agree on the verification results. Using only the yes-or-no answer it is hard to decide which tool is correct on disagreement. In those cases counterexamples and particularly proof certificates are not only essential to determine correctness, but they can also be used to debug correctness errors. Clearly, robustness and correctness are essential for the usability of tools. An incorrect verifier might for instance fail to detect errors in the implementation of a multiplier, i.e., reporting that the circuit is correct although it contains a bug.

Current research on algebraic multiplier verification focuses on efficiency and automation. However, we are not aware of research in the direction of automated fuzzing and debugging techniques for these tools that use AIGs as input. In automated reasoning fuzzing and delta debugging techniques have originally been applied for satisfiability modulo theories (SMT) [4], satisfiabilty (SAT) and quantified Boolean formulas (QBF) solvers [5], and interactive provers [19] and continue to be an essential part of the solver-developer tool-box [1,3,17,23,27,28,30]. For a general introduction to fuzzing we refer to the Fuzzing Book [35].

This paper gives an incentive towards investing effort in automated testing and debugging of automated reasoning tools, with the focus on hardware verification. We present our generation-based fuzzer MultAIGenFuzzer that produces valid multiplier circuits by randomly arranging modules of multiplier circuits. The goal of these benchmarks is to avoid overfitting of automated reasoning tools to those benchmark families that are currently publicly available and used, e.g., benchmarks generated by AMG [9], GenMul [22], and Multgen [32].

Additionally we present a mutation-based fuzzing tool AIGoFuzzing that applies small modifications to a given AIG, e.g., flipping signs. These mutations may or may not affect the correctness of the AIG. AIGoFuzzing is not tailored towards multiplier verification and may be applied to any given AIG, hence we consider it a general purpose mutation-based AIG fuzzer.

We expect that some of the fuzzing techniques deliver failure-inducing benchmarks that lead to solver crashes or incorrect results. Debugging failure-inducing benchmarks and locating the program error is typically a very time consuming task. Usually only a small part of the AIG raises the failing behavior, whereas the remainder of the graph is redundant with respect to the error. Reduction of tree-structured inputs is for example also discussed in [8]. In this paper we present a delta debugging tool AIGdd2 that allows to shrink failure-inducing inputs while preserving the failures. AIGdd2 is an updated re-implementation of AIGdd [2]; it still provides the same functionality as AIGdd, but has additional options to keep a multiplier-alike input/output signature of the AIG.

We evaluate the presented tools MultAIGenFuzzer, AIGoFuzzing, and AIGdd2 on the state-of-the-art multiplier verification tools DyPoSub [21] and AMulet2 [13]. Our experiments show that both tools have issues. For instance, we got benchmarks where DyPoSub miss-classifies the correctness of multipliers, i.e., claims that a multiplier is correct although it calculates $3 \cdot 3 = 13$. Our

tool AMulet2 did not produce wrong results, however it has several robustness issues. The presented approach allowed us to fix these issues in AMulet2.

The remainder of the paper is structured as follows, Sect. 2 discusses the input format AIG and introduces the used fuzzing and debugging techniques. In Sect. 3–5 we present our tools MultAIGenFuzzer, AIGoFuzzing, and AIGdd2. Section 6 discusses automated testing followed by our experiments in Sect. 7, before we conclude in Sect. 8 with a summary of learned lessons.

2 Fuzzing, Delta Debugging and AIGs

An *And-Inverter Graph* (AIG) [18] is a common representation of gate-level hardware circuits that is very efficient to handle and manipulate. An AIG is a directed acyclic graph that consists only of two-input nodes that represent logical conjunction. The edges may contain markings that indicate logical negations, i.e., inverters. The constant 0 in an AIG represents FALSE, and 1 represents TRUE. A small sample AIG is shown in Fig. 2.

Fuzzing is a technique for automated software testing. The original idea was to treat the program as a black-box and use random, invalid and unexpected inputs to detect failures and tool crashes, e.g., buffer overflows.

The origin of fuzz testing goes back to the 90's, where it was demonstrated that random inputs are able to detect many errors in UNIX command line programs [26]. Since then a variety of automated testing approaches and tools have been developed, such as ClusterFuzz by Google, or OneFuzz by Microsoft. The original testing approaches have recently been repeated on current UNIX systems [25] and still derived failure rates between 12%-19%. Especially pointer and array errors still seem to be present in state-of-the-art utilities.

Fuzzing approaches can be distinguished depending on whether they use existing input [35] or not. If a fuzzer reuses existing input, it is called *mutation-based* fuzzer, as it mutates the provided input by making small modifications. In contrast *generation-based* fuzzers generate input from scratch and hence do not depend on the existence of inputs (also called *seeds*).

A second differentiation relates to whether the fuzzer makes use of the structure of the tested program. A *white-box* fuzzer uses program analysis, e.g., symbolic execution [7] of either source-code or binaries, to systematically generate inputs with the goal to increase code coverage of the test program executing these inputs. On the opposite range of the spectrum, *black-box* fuzzing is completely unaware of the internal structure of the program under test. Hence, testing using black-box fuzzers is typically faster than white-box fuzzing and can easily be parallelized. However, black-box fuzzers may only trigger easy-to-reach bugs without ever reaching certain potentially harmful corner cases.

In this paper, we discuss two fuzzing tools. First MultAIGenFuzzer, a generation-based black-box fuzzer, generating multiplier circuits from scratch. Our second fuzzing tool AIGoFuzzing is a mutation-based black-box fuzzer. It has to be provided with an input seed, which is mutated without violating structural constraints, i.e., violating the input/output signature of multipliers.

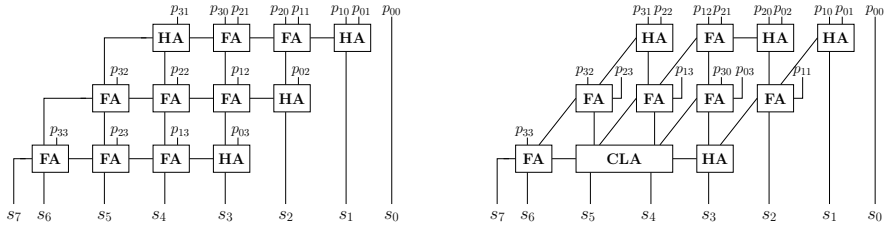


Fig. 1. 4-bit multipliers generated by MultAIGenFuzzer, where $p_{ij} = a_i b_j$. The left multiplier is generated using only full and half adders, the right multiplier uses a carry-lookahead module (CLA).

Delta debugging aims to reduce the manual workload of debugging software problems by minimizing failure-inducing inputs [33, 34, 36]. In a nutshell, delta debugging uses a (binary) search strategy to repeatedly remove smaller and smaller parts of the failure-inducing input until a minimal fix point is reached that triggers the same failure in the program. That is, our delta debugger AIGdd2 removes nodes from AIGs to narrow down the failure cause.

3 Generation-based Fuzzing for Multipliers

We consider n-bit multiplier circuits, with $2n$ inputs $a_{n-1} \dots a_0, b_{n-1} \dots b_0$ and $2n$ outputs $s_{2n-1} \dots s_0$, which compute $\sum_{i=0}^{2n-1} 2^i s_i = \left(\sum_{i=0}^{n-1} 2^i a_i\right)\left(\sum_{i=0}^{n-1} 2^i b_i\right)$.

In general multipliers are made of three components: (i) partial product generation (PPG), where the n^2 partial products $p_{ij} = a_i b_j$, $0 \leq i, j \leq n-1$ are generated; (ii) partial product accumulation (PPA), where the partial products are summed up to two layers, and (iii) a final-stage adder (FSA) [29].

For each component several hardware algorithms are available, which can be combined to derive a correct multiplier architecture, e.g., using a Booth-encoding for PPG, a Wallace-tree as PPA and a carry-lookahead adder as FSA. For example, the AMG [9] tool provides 22 components that can be combined to generate 192 different multiplier circuits (of arbitrary bit-width).

All these components have certain patterns and their number is limited. Thus hardware verification tools tend to be biased towards these architectures. For instance, our tool AMulet2 substitutes complex FSAs, such as carry-lookahead adders, with an equivalent ripple-carry adder. The correctness of the replacement is checked using SAT solving and the rewritten circuit is verified using computer algebra [14]. To apply adder substitution we implemented a heuristic algorithm that uses syntactic pattern matching to identify the FSA.

The purpose of our novel generation-based fuzzing tool MultAIGenFuzzer is to generate correct multipliers, where the individual components do not follow a specific pattern to avoid overfitting solvers to available benchmarks. Figure 1 shows abstract examples for multipliers generated by MultAIGenFuzzer.

We follow the natural approach of long multiplication: first all partial products are generated and uniquely assigned to $2n$ column-wise slices $S_0, \dots, S_{2n-1}$,

i.e., the partial product p_{ij} with $0 \leq i, j \leq n-1$ belongs to slice S_{i+j}. All partial products in the slices are added together to receive the output of the multiplication. Carries are propagated to the next bigger slices. Since addition is commutative the order of the partial products within a slice does not matter.

In MultAIGenFuzzer we generate all n^2 partial products p_{ij} using single and-gates. The integration of Booth encoding is part of future work. After the partial products are assigned to $2n$ slices, we repeatedly select two or three random elements of a randomly picked slice S_k and add them together using a half or full adder. The sum output bit of the half or full adder will be added to S_k, the carry will be added to S_{k+1}. Our tool supports two models with different density for half and full adders, which are selected randomly. We repeat these steps until all slices contain at maximum two elements.

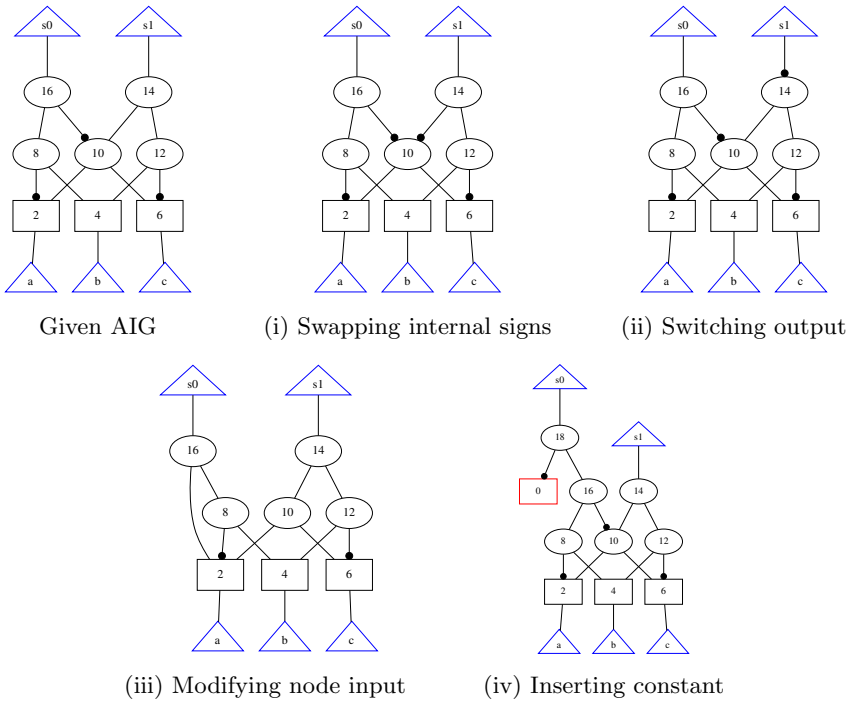
In the last step we generate a random FSA using a mixture of full and half adders and carry-lookahead addition. We traverse from the smallest slice S_0 to the largest slice S_{2n-1} and instantiate adder modules appropriate for the number of elements in a slice S_k. If the considered slice contains only a single element, this element becomes the k-th output of the multiplier. If a slice contains two elements we use a half adder to sum up the elements. The sum output is the k-th output bit of the multiplier circuit, the carry is inserted into the next slice S_{k+1}. In the case a slice S_k contains three elements, e.g., two elements from the partial product accumulation and one carry from the FSA module of slice S_{k-1}, we further randomly choose between a full adder or a carry-lookahead adder. We randomly select the carry-lookahead module with a probability of 2/3.

If a full adder is selected, summing up the three elements follows the same procedure as for a half adder. If a carry-lookahead adder is chosen, we first count the number of coherent slices $S_{k+1}, \ldots, S_{2n-1}$ that contain two elements, as this will be the maximum size max of the carry-lookahead adder. We select a random number in $[1, max]$ to choose the actual size of the carry-lookahead adder.

The sum outputs of a carry-lookahead adder are generated using XOR gates. The carries can be computed recursively or iteratively. We have decided to use three modes in MultAIGenFuzzer, which are selected randomly. In the first mode all carries of the carry lookahead adder are computed recursively. In a second mode all carries are computed iteratively. In the third mode we mix between the first two options, i.e., we randomly select for each carry whether it is generated recursively or iteratively.

4 Mutation-based Fuzzing for AIGs

In contrast to MultAIGenFuzzer, presented in the previous section, our second fuzzer AIGoFuzzing is a mutation-based fuzzing tool and requires existing seeds. The purpose of AIGoFuzzing is to make small mutations in a given AIG that may or may not change the specification of the graph. Our fuzzer AIGoFuzzing is not specifically designed for multiplier verification and can be used on any given general-purpose AIG.



Given AIG (i) Swapping internal signs (ii) Switching output

(iii) Modifying node input (iv) Inserting constant

Fig. 2. A given AIG, that is mutated using one of the four available muations (i)–(iv)

We currently support four classes of mutations, which can be selected optionally. The provided mutations are (i) swapping internal signs, (ii) switching an output signal, (iii) modifying the input of a node, and (iv) inserting constants 0 and 1. An example is given in Fig. 2. The location of the mutation is chosen randomly and if not specified otherwise AIGoFuzzing only executes single mutations. Applying multiple modifications is possible via the options.

Swapping internal signs may happen in two ways. Either the sign of a single randomly chosen edge is flipped, or a complete node is flipped by switching the signs of all outgoing edges. Switching an output signal is actually a special case of flipping a complete node, yet we have decided to include this as a separate mutation technique, as AIGs that encode miters consist only of one output bit. Modifying the sign of the output bit allows us to negate the specification of the whole AIG. In all cases, flipping the signs usually changes the behavior of the input AIG, i.e., in our setting correct multipliers will become incorrect. We will revisit this claim later when discussing Tbl. 3 in the experimental evaluation.

In the third mutation technique we modify the input of an AIG node. We randomly select a parent node, one of its children that is replaced, and a topologically smaller node that will become the new input. It is important that we choose a new child node that is topologically smaller than the parent to prevent the formation of cycles in the generated AIG. Of course we do expect that most of the time modifying the input of an AIG affects the correctness of the AIG.

The fourth mutation technique injects constants 0 and 1 into the AIG. First we randomly select which constant is inserted and then whether it will be added as an input of a random node using logical conjunction or disjunction. This yields four possible combinations (0, conjunction), (0, disjunction), (1, conjunction), and (1, disjunction). Since $v \wedge 1 = v \vee 0 = v$ for any node v, integrating the constant 0 using disjunction, and 1 using conjunction will not affect the correctness of the AIG. On the other hand $v \wedge 0 = 0$ and $v \vee 1 = 1$. Thus the combinations (0, conjunction) and (1, disjunction) will change the behavior. This can be seen in Tbl. 3 in the experimental evaluation that we have a 50:50 ratio on the correctness of the tested multipliers.

5 Delta Debugging with Slices

AIGDD2 is a delta debugging tool that is designed to minimize failure-inducing AIGs, while preserving the errors. AIGDD2 is a re-implementation of AIGDD [2], which has been developed for model checking purposes, whereas this paper focuses on multiplier verification tools. In particular we extend the functionality of AIGDD by adding options that limit structural changes to the AIG, e.g., to maintain the signature of multiplier circuits. Furthermore we include a slicing based delta debugging approach that allows us to shrink the bit-width of multipliers.

The goal of AIGDD2 is to automatically reduce the size of failure-inducing AIGs until a fixed point is reached. AIGDD2 reads and executes the failure-inducing AIG on the buggy program. The exit code of the faulty program is stored and will be used as golden reference value.

First AIGDD2 slices the given AIG by removing the most significant inputs and outputs. We set these bits to 0 and propagate the constant, e.g, for node v with inputs w and 0, we deduce $w \wedge 0 = v = 0$. The precise number of inputs and outputs that are set to 0 in each iteration can be specified via the options, e.g., for shrinking multipliers we always remove two outputs and two inputs. We repeat slicing as long as the shrunken AIG triggers the failing behavior.

Then AIGDD2 uses a binary-search based approach to further shrink the size of the sliced AIG. We start by setting the first half of AIG nodes to the constant 0, which is propagated. We then test whether the resulting AIG still triggers the failing behavior in the buggy program. If so, we proceed with the second half of the AIG nodes. If the buggy program returns a different exit code, we try again with setting the first half of AIG nodes to the constant 1. If setting the first half of AIG nodes to a constant 0 or 1 does not trigger the failure of the tested program, we split the first half of the AIG nodes into two parts and repeat the steps above. We repeatedly narrow down the search space until we have set individual AIG nodes to 0 resp. 1.

At this point we want to emphasize that it is not guaranteed that AIGDD2 generates the smallest possible failure-inducing input. However, while debugging errors it is most of the time not needed to generate a minimal example. The desire is to generate small inspectable inputs that can be used for efficient debugging.

6 Fuzzing, Tests and Proofs

We evaluate our presented fuzzing and debugging approaches on the most recent multiplier verification tools DyPoSub [21] and AMulet2 [13], where we are the authors of the latter. Both tools apply algebraic reasoning to decide on the correctness of the given multiplier. The specification of the circuit, modeled as a polynomial is reduced by the polynomial representation induced by the AIG with regard to a reverse topological variable ordering. The circuit is correct if and only if the final result is zero.

In our tool AMulet2 we replace complex FSAs, which are a bottleneck for algebraic reasoning [14], with a simple ripple-carry adder that is easy for algebraic reasoning. The rewritten circuit is verified in AMulet2 using a static variable ordering. We use version AMulet2.1 [10] for our experimental evaluation.

The tool DyPoSub applies a dynamic variable ordering attempt, i.e., after each reduction step the change in the size of the current intermediate reduction result is investigated. If the growth is above a certain threshold, the reduction step is undone and a different variable is considered for reduction. This approach aims to guarantee that the intermediate reduction results do not explode.

We use our presented fuzzing and delta debugging techniques in two ways. First, we investigate the *robustness* of the verification tools. More precisely we generate multiplier circuits using MultAIGenFuzzer that will be mutated using AIGoFuzzing. We run these benchmarks on AMulet2.1 and DyPoSub to detect critical software failures and crashes. With these results we fixed several issues in our tool, also with the help of AIGdd2. The resulting more robust version AMulet2.2 is released with this paper.

Secondly, we aim to deduce the *correctness* of the verification tools using differential testing [24], see Fig. 3. We run both tools AMulet2.2 and DyPoSub on the generated benchmarks that may or may not be correct. If both solvers agree on the correctness of the given circuit, we label the multiplier accordingly and conclude that the tools are correct on these benchmarks. This corresponds to the green boxes in Fig. 3 (top & bottom of the middle column).

However, since we only have two solvers available we automatically have a tie when the solvers disagree on the verification results. Obviously one of the tools has a correctness issue, cf., the red boxes in Fig. 3. We use the following terminology as common in logical reasoning and call a tool *unsound* whenever it decides that a given buggy multiplier is correct. If a correct multiplier is classified as buggy we call the solver *incomplete*.

Since we do not have a golden verification tool available, it is hard to determine which solver is correct and which is buggy without additional actions. The tool DyPoSub only provides a yes-or-no answer as a verification result, whereas AMulet2 additionally provides certificates. In the case AMulet2 returns "correct circuit" it generates on-the-fly a proof certificate in the practical algebraic calculus (PAC) [15] that independently monitors the reduction steps using a series of simple polynomial operations. The proof certificates can be checked using the proof checkers Pacheck 2.0 or Pastèque 2.0 [15], where the latter is itself certified as it is derived using Isabelle/HOL.

If AMulet2 considers the given circuit to be incorrect, we are able to provide at least one concrete counterexample from the final non-zero reduction result, i.e., we provide a concrete assignment of inputs for which the multiplier produces a wrong result. Due to the reverse topological variable ordering the final reduction polynomial consists only of input variables of the multiplier. We choose the smallest monomial of the polynomial and set all occurring variables to 1, and all other input variables to 0. Hence, the polynomial evaluates to the non-zero coefficient of the selected monomial. In the case we have multiple monomials of minimial size available, we are able to provide several counterexamples. These counterexamples can be used to test the correctness of the multiplier by simulating the outputs, e.g., using AIGSim [2]. AIGSim computes the output of the AIG for the provided input model.

So in these cases where AMulet2 and DyPoSub disagree on the correctness of a circuit, proofs and testing of counterexamples allow us to have a high level of confidence in the verification results of AMulet2. If both tools provide different yes-or-no answers we use the additional tests and proofs by AMulet2. If these tests and proofs are successfully checked by independent simulators and proof checkers, we use the verification result of AMulet2. If the proof certificates and counterexamples are not valid we use the verification result of DyPoSub and classify AMulet2 as unsound or incompolete.

This tool debugging flow also shows the importance of tests and proofs and how developers of tools that do not provide certificates can make use of counterexamples and proofs of a second tool to debug correctness issues in their tool. Moreover in the case that DyPoSub would provide counter examples too, providing proofs in AMulet2 becomes superfluous and we could make use of their counter examples to check for unsound errors in AMulet2.

Theoretically we would also need to consider the cases in Fig. 3, where both tools return the same yes-or-no answer, but checking the proofs and counterexamples of AMulet2 fails. However, we have not seen this case in practice, i.e., whenever booth tools returned the same yes-or-no answer, the proofs and counterexamples supported this decision, and thus decided to not include them in Fig. 3.

7 Experiments

Our experiments use an Intel Xeon E5-2620 v4 CPU at 2.10 GHz (with turbo-mode disabled) with a memory limit of 128 GB. The time is listed in seconds (wall-clock time). We compare our tool AMulet2 to related work DyPoSub, and consider two versions of AMulet2: (i) AMulet2.1 as a slightly updated version of AMulet2.0 [13] that is currently available on GitHub [10] and (ii) AMulet2.2, a fixed version of AMulet2.1 that will be released together with this paper. For DyPoSub we use the version described in [21], but, as the tool DyPoSub is not yet publicly available, we use a binary kindly provided by the authors. In all our experiments the time limit has been set to 300 seconds and the memory limit has been set to 16 GB.

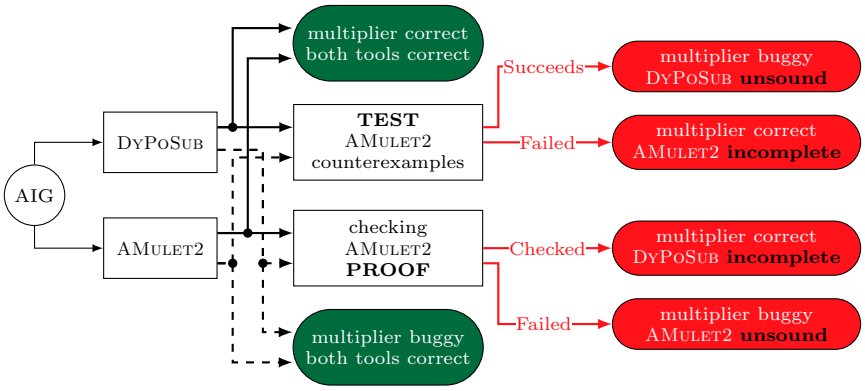


Fig. 3. Determining the correctness of tools using tests and proofs

In our experimental evaluation we aim to investigate the robustness and correctness of the aforementioned multiplier verification tools using benchmarks that are generated by (i) MultAIGenFuzzer, (ii) AIGoFuzzing on available benchmarks from the AMG-benchmarks [9], and (iii) AIGoFuzzing on benchmarks generated by MultAIGenFuzzer. Additionally we show the effect of AIGdd2 on reducing the size of failure-inducing inputs. All our experimental data and source code is available at [12].

For each tool we split the results into ✔and ✗. The ✔-columns show those results where a tool is correct, i.e., where it provides a yes-answer for a correct multiplier (👍), or classifies a faulty multiplier as incorrect (👎). The ✗-columns show failures and miss-classifications, and we use the following symbols to differentiate different types of failures as well as correct results:

✔	✗
👍 correct multiplier marked as correct	👍 buggy multiplier marked as correct (unsound)
👎 buggy multiplier marked as incorrect	👎 cor. multiplier marked as incorrect (incomplete)
	↯ static ordering not topological (AMulet2 only)
	↯ segmentation fault
	⌛ exceeding the time limit

7.1 MultAIGenFuzzer generates Multipliers

The purpose of these experiments is to investigate whether the considered tools have been overfitted to available multiplier benchmarks, e.g. [9, 22, 32]. We use MultAIGenFuzzer to generate correct multiplier circuits, where the internal structure of the multipliers is shuffled.

In our first experiment that is shown in Tbl. 1, we generate 2 000 random benchmarks with an input bit-width $n \in \{4, 8, 16, 32, 64\}$. In this experiment we exclude the usage of carry-lookahead modules within the circuits, i.e., all benchmarks exist only of full and half adders that are arranged in a random, but topologically correct order. On a first glance this seems to be a rather easy

Table 1. MultAIGenFuzzer benchmarks without carry-lookahead modules.

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
n	#	👍	👎	👍	👎	↯	ϟ	⧗	👍	👎	👍	👎	↯	ϟ	⧗	👍	👎	👍	👎	ϟ	⧗
4	401	401	0	0	0	0	0	0	401	0	0	0	0	0	0	401	0	0	0	0	0
8	414	414	0	0	0	0	0	0	414	0	0	0	0	0	0	414	0	0	0	0	0
16	385	385	0	0	0	0	0	0	385	0	0	0	0	0	0	385	0	0	0	0	0
32	406	406	0	0	0	0	0	0	406	0	0	0	0	0	0	406	0	0	0	0	0
64	394	394	0	0	0	0	0	0	394	0	0	0	0	0	0	394	0	0	0	0	0
	2 000	2 000	0	0	0	0	0	0	2 000	0	0	0	0	0	0	2 000	0	0	0	0	0

fuzzing setup, however a large number of such benchmarks has been submitted to the SAT Race 2019 [16] and the results show that shuffling the addition order of partial products is hard for SAT solving. We now aim to investigate these benchmarks on algebraic solving techniques.

It can be seen that all tools are able to solve all benchmarks within the given time limit and correctly return that the given benchmarks are indeed correct multipliers. None of the tools has crashes nor returns unsound or incomplete results. Hence, we conclude that these algebraic verification tools are robust with respect to shuffling the addition order of the partial products.

In our second experiment we use MultAIGenFuzzer to generate correct multipliers. For these benchmarks we enable the usage of carry-lookahead modules in the FSA and we want to investigate whether the solvers have been overfitted to certain FSAs. We generate 2 000 random benchmarks with an input bit-width $n \in \{4, 6, 8, 12, 16, 32, 48, 64\}$. We include more input bit-widths than in the first experiment to allow a more detailed discussion. The results can be seen in Tbl. 2, which is organized in the same fashion as Tbl. 1.

The results show that DyPoSub is able to correctly verify all benchmarks within the given time limit. AMulet2.1 as well as the new release AMulet2.2 exceed the time limit on more than 50% of the benchmarks. The number of timeouts increases as the input bit-width increases and AMulet2 is not able to verify any of the 64-bit benchmarks. The reason is that AMulet2 aims to apply adder substitution using syntactic pattern matching before algebraic verification of the rewritten circuit. Since the adder substitution algorithm is not targeted towards mixed FSAs, adder substitution fails in these cases, i.e., algebraic reduction is applied on the original circuit. Hence we are able to conclude that the adder substitution step in AMulet2 is overfitted to known patterns and struggles with random complex inputs.

7.2 Using AIGoFuzzing to Mutate AIGs

In the following experiments we use small 4-bit multipliers that are generated by AMG [9] and use AIGoFuzzing to apply single mutations.

Table 2. MultAIGenFuzzer benchmarks with carry-lookahead modules.

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
n	#	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	ϟ	⧖
4	247	247	0	0	0	0	0	0	247	0	0	0	0	0	0	247	0	0	0	0	0
6	221	218	0	0	0	0	0	3	217	0	0	0	0	0	4	221	0	0	0	0	0
8	251	222	0	0	0	0	0	29	225	0	0	0	0	0	26	251	0	0	0	0	0
12	252	141	0	0	0	0	0	111	141	0	0	0	0	0	111	252	0	0	0	0	0
16	249	65	0	0	0	0	0	184	65	0	0	0	0	0	184	249	0	0	0	0	0
32	240	10	0	0	0	0	0	230	10	0	0	0	0	0	230	240	0	0	0	0	0
48	257	1	0	0	0	0	0	256	1	0	0	0	0	0	256	257	0	0	0	0	0
64	283	0	0	0	0	0	0	283	0	0	0	0	0	0	283	283	0	0	0	0	0
	2 000	904	0	0	0	0	0	1 096	906	0	0	0	0	0	1 094	2 000	0	0	0	0	0

First, we select the multiplier circuit "sp-ar-rc-4.aig", that uses a simple partial product generation, i.e., and-gates, which are accumulated in an array structure. The final stage adder is a ripple-carry adder. The AIG is shown in Fig. 4 in the appendix, This architecture consists purely of full and half adders arranged in a grid-like pattern and thus is considered to be a simple multiplier in related work. This circuit is given to AIGoFuzzing where randomly one of four presented mutation techniques is applied.

The results can be seen in Tbl. 3, where the first column refers to the mutation. AIGoFuzzing is able to apply four different kinds of mutations: changing the sign of a random internal edge or node ("intsign"), swapping a random output signal ("outsign"), replacing the input of an internal node ("inptrpl") and integrating a constant ("const+").

We see that changing the internal sign leads to problems with finding an appropriate variable ordering in AMulet2.1. We apply delta debugging on these benchmarks, cf., Tbl. 7 and are able to fix all issues in AMulet2.2. Swapping the sign of an output bit did not produce any crashes nor failures on any of the tools. Modifying the input of an internal node or integrating a constant leads to problems with finding a correct topological variable ordering in AMulet2.1 as well as related segmentation faults. More specifically, integrating a constant leads to crashes of AMulet2.1 on around 50% of the benchmarks.

The tool DyPoSub is robust and does not crash nor does it exceed the time limit on any of the benchmarks. However, in the row "inptrpl" DyPoSub reports that 20 circuits are correct, whereas AMulet2.2 only reports 14 circuits to be correct. That is, on 14 circuits both tools return "correct" and on 6 benchmarks DyPoSub classifies the multiplier to be correct whereas AMulet2 considers the circuit to be buggy. For these 6 circuits we require additional information to resort the tie. We use the counterexamples that are generated by AMulet2.2 and simulate them on the input AIGs. All counterexamples are found to be valid. To be 100% sure of the claim that DyPoSub is unsound, we additionally

Table 3. AIGoFuzzing "sp-ar-rc-4" multipliers

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]						
		✔		✗					✔		✗					✔		✗				
mutation	#	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	ϟ	⧖
intsign	505	31	442	0	0	32	0	0	31	474	0	0	0	0	0	31	474	0	0	0	0
outsign	524	0	524	0	0	0	0	0	0	524	0	0	0	0	0	0	524	0	0	0	0
inptrpl	475	14	344	0	0	59	58	0	14	461	0	0	0	0	0	14	455	6	0	0	0
const+	496	232	0	0	0	12	252	0	252	244	0	0	0	0	0	252	244	0	0	0	0
	2 000	277	1 309	0	0	103	311	0	297	1 703	0	0	0	0	0	297	1 702	**6**	0	0	0

generated miters for checking the equivalence of the AIGs, which we gave to a SAT solver. The SAT solver returned SAT, thus the AIGs in question are indeed buggy. Details on the simulations, miters, and results of the SAT solver are included in our experimental data [12]. We select one of the multipliers that causes the unsound-failure. With the help of our delta debugger AIGdd2 we are able to shrink the failure-inducing AIG by 86% from 128 nodes to 18 nodes. The minimized AIG and its simulation can be seen in Fig. 6 in the appendix. We have provided our findings to the authors of DyPoSub and they were able to locate errors in their tool.

Interestingly, not all mutations where we swapped internal signs and modified the input of nodes lead to incorrect multipliers, which we would have expected. Around 5% of the multipliers were we swap signs, and 2% of the multipliers where we modify inputs of nodes remain correct.

However, we see that only the fourth mutation technique "const+" keeps roughly a 50:50 balance between correct and incorrect multipliers, all other mutations are highly unbalanced. Developing semantics-preserving mutation techniques that preserve correctness of AIGs is part of future work.

We repeated these experimetns of Tbl. 3 with a more complex multiplier "bp-ba-lf-4.aig" (Fig. 5 in the appendix). This circuit uses a Booth-encoding to generate the partial products, which are then accumulated using a redundant binary addition tree and the FSA is a complex Ladner-Fischer adder. This circuit is modified by AIGoFuzzing.

These results are shown in Table. 4 and it turns out that AMulet2.1 crashes on around 15% of the benchmarks and does not find a correct variable ordering in around 8% of the cases. However, all of these issues have been resolved in AMulet2.2. Furthermore, DyPoSub claims that almost all multipliers are incorrect, verifies one circuit and crashes on a second one. On the other hand, AMulet2.2 reports "correct multiplier" on 260 multipliers.

Accordingly, we check the provided proof certificates by AMulet2.2 as support of the verification result. For all multipliers that have been classified as correct by AMulet2.2, the proof certificates are correctly checked by the certified proof checker Pastèque 2.0 [15]. Hence, DyPoSub is incomplete on these 258 benchmarks.

Table 4. AIGoFuzzing "bp-ba-lf-4" multipliers

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
mutation	#	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	ϟ	⧖
intsign	529	23	453	0	0	53	0	0	23	506	0	0	0	0	0	0	506	0	23	0	0
outsign	476	0	476	0	0	0	0	0	0	476	0	0	0	0	0	0	476	0	0	0	0
inprpl	544	9	383	0	0	71	81	0	10	534	0	0	0	0	0	0	534	0	9	1	0
const+	451	193	0	0	0	32	226	0	227	224	0	0	0	0	0	1	224	0	226	0	0
	2 000	225	1 312	0	0	156	307	0	260	1 740	0	0	0	0	0	1	1 740	0	**258**	1	0

Table 5. MultAIGenFuzzer multipliers + single mutations of AIGoFuzzing

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
n	#	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	↯	ϟ	⧖	👍	👎	👍	👎	ϟ	⧖
4	403	61	253	0	0	19	70	0	66	337	0	0	0	0	0	66	337	0	0	0	0
6	398	41	235	0	0	15	61	46	43	334	0	0	0	0	21	45	352	0	0	0	1
8	426	47	160	0	0	22	93	104	49	256	0	0	0	0	121	55	295	0	0	0	76
12	386	32	112	0	0	6	75	161	31	169	0	0	0	0	185	44	204	0	0	0	138
16	387	38	98	0	0	14	74	163	40	160	0	0	0	0	187	54	184	0	0	0	149
	2 000	219	858	0	0	76	373	474	229	1 257	0	0	0	0	514	264	1 372	0	0	0	364

7.3 Combining Generation- and Mutation-Based Fuzzing

In these experiments we combine our presented fuzzing tools and generate circuits using MultAIGenFuzzer that are modified using AIGoFuzzing.

In the experiment that can be seen in Tbl. 5 we generate benchmarks with an input size $n \in \{4, 6, 8, 12, 16\}$ and apply a single random mutation, i.e., any of the four available types.

We see that AMulet2.1 crashes on $\sim 18\%$ of these benchmarks and has an error in finding a topological variable order in 76 cases. These issues are fixed in AMulet2.2. DyPoSub is robust and correct on these benchmarks. For all tools we see that the larger the input size the more likely is it to exceed the time limit, i.e., we produce hard benchmarks for these solvers.

In the experiment in Tbl. 6 we apply a number of random mutations on a 4-bit multiplier. After the experiment in Tbl. 6 with $n = 4$ we tried various other input sizes and it turns out that already bit-width 6 leads to time outs for a large number of benchmarks. The corresponding Tbl. 8 for $n = 6$ can be found in the appendix. It also shows that increasing the number of mutations also increases the number of crashes in AMulet2.1. Almost 75% of the multipliers either lead to a segmentation fault or an error in the variable ordering. Again, we were able to fix these issues in AMulet2.2. Finally, note, that AMulet2.2 and DyPoSub do not exceed the time limit on any of these benchmarks and consistently produce the same verification result.

Table 6. MultAIGenFuzzer multipliers +mutations of AIGoFuzzing

		AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
#mut	#	👍	👎	👍	👎	⚡̸	↯	⧖	👍	👎	👍	👎	⚡̸	↯	⧖	👍	👎	👍	👎	↯	⧖
2	191	5	123	0	0	12	51	0	5	186	0	0	0	0	0	5	186	0	0	0	0
4	233	0	120	0	0	18	95	0	0	233	0	0	0	0	0	0	233	0	0	0	0
6	207	0	75	0	0	15	117	0	0	207	0	0	0	0	0	0	207	0	0	0	0
8	223	0	59	0	0	9	155	0	0	223	0	0	0	0	0	0	223	0	0	0	0
10	180	0	42	0	0	3	135	0	0	180	0	0	0	0	0	0	180	0	0	0	0
12	194	0	30	0	0	4	160	0	0	194	0	0	0	0	0	0	194	0	0	0	0
14	181	0	19	0	0	1	161	0	0	181	0	0	0	0	0	0	181	0	0	0	0
16	203	0	18	0	0	4	181	0	0	203	0	0	0	0	0	0	203	0	0	0	0
18	190	0	19	0	0	1	170	0	0	190	0	0	0	0	0	0	190	0	0	0	0
20	198	0	11	0	0	2	185	0	0	198	0	0	0	0	0	0	197	0	0	1	0
2	000	5	516	0	0	69	1	410	0	5	1	995	0	0	0	0	0	5	1	994	0	0	1	0

7.4 Using AIGdd2 to Minimize Failure-Inducing Inputs

Tables 3–6 show that AMulet2.1 is very sensitive with respect to mutations. It produced many crashes and errors while finding an appropriate variable ordering. In order to reduce time consuming debugging during fixing these issues it is beneficial to automatically shrink failure-inducing inputs through delta-debugging. In order to determine the effectiveness of delta-debugging in this context we applied our delta-debugger AIGdd2 with and without slicing on all benchmarks that cause a failure in the variable ordering, i.e., all benchmarks in columns "⚡̸" of AMulet2.1 in Tbls. 3–6. The results are shown in Tbl. 7.

The table is split into four segments, one for each original table. The rows in each segment are organized in the same fashion as in the original tables, e.g., the first segment considers Tbl. 3, where each row indicates the used mutation type. We also list the number of benchmarks that exceed the time limit of 300 seconds provided for delta debugging (⧖), benchmarks where delta debugging fails (✗), and where AIGdd2 is able to shrink the input (✔).

It can be seen that AIGdd2 is able to find smaller failure-inducing benchmarks for the multipliers where either internal signs have been swapped or the input of a node has been modified. The columns in block "✔" show the number of benchmarks ("#"), the minimum percentage of nodes that is removed ("min"), the largest percentage that is removed ("max") and the average size removal ("avg"). We see that for the failure-inducing benchmarks of Tbl. 3, AIGdd2 is able to reduce the size by half on average. For the failure-inducing benchmarks of Tbl. 4, AIGdd2 is able to reduce the size by two third.

For those benchmarks, where we integrate constants, AIGdd2 is not able to find smaller inputs that are failure-inducing, cf., column ✗. The reason is in the functionality of AIGdd2. AIGdd2 sets large parts of the AIG to constants and

propagates these values, i.e., the internal constant is propagated to the outputs of the AIG. Benchmarks of this kind do not produce a failure in AMulet2.1. A closer inspection of AMulet2.1 showed that we considered multipliers with constant outputs and inputs, but did not consider constants as inputs of internal nodes. Overall we see that slicing-based delta debugging finds smaller failure-inducing benchmarks.

For the benchmarks of Tbl. 5 we see that AIGdd2 several times exceeds the time limit of 300 seconds for multipliers with an input bit-width $n > 4$. We have repeated the experiment with a time limit of 3 600 seconds, which shows the same outcome. For those multipliers which could be minimized, AIGdd2 is able to find failure-inducing multipliers that are 96.7% smaller.

For the benchmarks of Tbl. 6 it can be seen that AIGdd2 does not find smaller benchmarks on 10 out of 69 test cases. In all other cases the percentage of removal is between 44.5%–84.2%.

7.5 Summary

Our experiments have shown that AMulet2 is overfitted to existing FSAs. Especially those MultAIGenFuzzer-benchmarks, where the FSA is not a clean ripple-carry adder or a pure carry-lookahead adder, but a random combination of both adders are very hard for AMulet2. The tool DyPoSub is robust and correct on these benchmarks.

We encountered that AMulet2.1 has several issues regarding its robustness on random mutated input circuits. Especially the variable ordering algorithm has severe issues. Delta debugging these failure-inducing inputs was extremely helpful to find the errors in AMulet2.1, which are fixed in the more robust version AMulet2.2.

Using mutation-based fuzzing we have found multipliers where DyPoSub is unsound and others where it is incomplete, i.e., produces incorrect verification results. Since AMulet2 produces checkable proof certificates and counterexamples in addition to a simple yes-or-no answer, it adds another layer of confidence in the verification result and we did not encounter any soundness and completeness failures in AMulet2.

8 Conclusion

Software is only as good as its robustness and correctness. A software that frequently crashes is as undesirable as software that returns incorrect results. In this paper we have evaluated the robustness and correctness of automated multiplier verification tools that use algebraic reasoning. Our presented generation- and mutation-based fuzzing techniques together with the presented delta debugger allow us to detect and debug issues in these tools. Our experiments show that both of the considered tools have critical defects, i.e., DyPoSub [21] is unsound and incomplete, and our tool AMulet2 [13] crashes several times. We fixed the issues in the new release AMulet2.2.

Table 7. Reducing the size of failure-inducing benchmarks with AIGDD2

		-slicing						+slicing					
	mutation	⧖	✗	✔				⧖	✗	✔			
				#	min(%)	max(%)	avg(%)			#	min(%)	max(%)	avg(%)
Tbl. 3	intsign	0	0	32	49.2	53.1	51.7	0	0	32	49.2	68.0	56.5
	inptrpl	0	0	59	48.4	58.3	52.5	0	0	59	48.4	82.0	58.4
	const+	0	12	0	-	-	-	0	12	0	-	-	-
	mutation	⧖	✗	✔				⧖	✗	✔			
				#	min(%)	max(%)	avg(%)			#	min(%)	max(%)	avg(%)
Tbl. 4	intsign	0	0	53	60.2	69.3	65.5	0	0	53	60.2	86.5	67.4
	inptrpl	0	0	71	58.8	73.4	66.1	0	0	71	58.8	88.5	68.5
	const+	0	32	0	-	-	-	0	32	0	-	-	-
	n	⧖	✗	✔				⧖	✗	✔			
				#	min(%)	max(%)	avg(%)			#	min(%)	max(%)	avg(%)
Tbl. 5	4	0	4	15	43.5	63.4	53.0	0	4	15	43.5	63.4	53.0
	6	12	3	0	-	-	-	7	3	5	74.6	89.7	81.8
	8	20	3	0	-	-	-	13	2	7	87.7	94.3	91.0
	12	6	0	0	-	-	-	5	0	1	96.7	96.7	96.7
	16	11	3	0	-	-	-	12	2	0	-	-	-
	#mut	⧖	✗	✔				⧖	✗	✔			
				#	min(%)	max(%)	avg(%)			#	min(%)	max(%)	avg(%)
Tbl. 6	2	0	3	9	47.0	59.8	53.8	0	3	9	47.0	70.3	60.4
	4	0	4	14	44.5	71.3	56.5	0	4	14	44.5	71.3	60.0
	6	0	1	14	43.1	67.3	56.1	0	1	14	45.0	72.0	61.4
	8	0	1	8	47.8	67.2	58.0	0	1	8	54.1	68.8	64.0
	10	0	0	3	55.8	65.0	60.1	0	0	3	55.8	74.8	63.4
	12	0	0	4	54.2	68.5	61.2	0	0	4	58.9	84.2	73.7
	14	0	0	1	56.6	56.6	56.6	0	0	1	56.6	56.6	56.6
	16	0	0	4	52.4	64.2	55.9	0	0	4	52.4	70.6	60.0
	18	0	1	0	-	-	-	0	1	0	-	-	-
	20	0	0	2	56.7	56.8	56.8	0	0	2	56.7	67.8	62.3

We conclude with observations which we believe to generalize for testing and debugging other verification tools. First, randomly shuffling the structure of available inputs helps to avoid overfitting to known benchmark families. Second, even only small mutations are able to reveal defects in an efficient way. Third, verification tools need to produce proof certificates and models in addition to a yes-or-no answer to prevent false results. Fourth, shrinking failure-inducing inputs using delta debugging is extremely helpful for zooming in on defects.

In the future we want to include dedicated sophisticated semantic preserving mutations in AIGOFUZZING in order to achieve a more balanced ratio between correct and incorrect benchmarks.

References

1. C. Artho, A. Biere, and M. Seidl. Model-based testing for verification back-ends. In *TAP@STAF*, volume 7942 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2013.
2. A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 And Beyond. Technical report, FMV Reports Series, JKU Linz, Austria, 2011.
3. D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh. StringFuzz: A fuzzer for string solvers. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 45–51. Springer, 2018.
4. R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *SMT Workshop*, SMT '09, page 1–5, New York, NY, USA, 2009. Association for Computing Machinery.
5. R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. In *SAT*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.
6. M. J. Ciesielski, T. Su, A. Yasin, and C. Yu. Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE TCAD*, pages 1–1, 2019. Early acces.
7. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.
8. S. Herfert, J. Patra, and M. Pradel. Automatically reducing tree-structured test inputs. In *ASE*, pages 861–871. IEEE Computer Society, 2017.
9. N. Homma, Y. Watanabe, T. Aoki, and T. Higuchi. Formal Design of Arithmetic Circuits Based on Arithmetic Description Language. *IEICE Transactions*, 89-A(12):3500–3509, 2006.
10. D. Kaufmann. Amulet 2.1. https://github.com/d-kfmnn/amulet. SHA 8e1838fa4c6d80091869407c44519e2771694b21.
11. D. Kaufmann. *Formal Verification of Multiplier Circuits using Computer Algebra*. PhD thesis, Computer Science, Johannes Kepler University Linz, 2020.
12. D. Kaufmann. Artifact for fuzzing and delta-debugging and-inverter graph verification tools. http://fmv.jku.at/aigfuzzing_artifact, 2022.
13. D. Kaufmann and A. Biere. Amulet 2.0 for verifying multiplier circuits. In *TACAS (2)*, volume 12652 of *LNCS*, pages 357–364. Springer, 2021.
14. D. Kaufmann, A. Biere, and M. Kauers. Verifying Large Multipliers by Combining SAT and Computer Algebra. In *FMCAD 2019*, pages 28–36. IEEE, 2019.
15. D. Kaufmann, M. Fleury, A. Biere, and M. Kauers. Practical Algebraic Calculus and Nullstellensatz with the Checkers Pacheck and Pastèque and Nuss-Checker. *Formal Methods Syst. Des.*, 2022. To appear.
16. D. Kaufmann, M. Kauers, A. Biere, and D. Cok. Arithmetic Verification Problems Submitted to the SAT Race 2019. In *SAT Race 2019*, volume B-2019-1 of *Dep. of Computer Science Report Series B*, page 49. University of Helsinki, 2019.
17. G. Kremer, A. Niemetz, and M. Preiner. ddSMT 2.0: better delta debugging for the SMT-LIBv2 language and friends. In *CAV (2)*, volume 12760 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2021.
18. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE TCAD*, 21(12):1377–1394, 2002.
19. L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA):181:1–181:29, 2019.

20. A. Mahzoon, D. Große, and R. Drechsler. RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers. In *DAC 2019*, pages 185:1–185:6. ACM, 2019.
21. A. Mahzoon, D. Große, C. Scholl, and R. Drechsler. Towards formal verification of optimized and industrial multipliers. In *DATE*, pages 544–549. IEEE, 2020.
22. A. Mahzoon, D. Große, and R. Drechsler. Multiplier Generator GenMul. http://www.sca-verification.org/, 2019.
23. M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/SIGSOFT FSE*, pages 701–712. ACM, 2020.
24. W. M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.
25. B. Miller, M. Zhang, and E. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Trans. Softw. Eng.*, pages 1–1, 2020. early acces.
26. B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.
27. A. Niemetz and A. Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In R. Bruttomesso and A. Griggio, editors, *Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013*, pages 36–45, 2013.
28. A. Niemetz, M. Preiner, and A. Biere. Model-based API testing for SMT solvers. In *SMT*, volume 1889 of *CEUR Workshop Proceedings*, pages 3–14. CEUR-WS.org, 2017.
29. B. Parhami. *Computer Arithmetic - Algorithms and Hardware designs*. Oxford University Press, 2000.
30. J. Scott, T. Sudula, H. Rehman, F. Mora, and V. Ganesh. Banditfuzz: Fuzzing SMT solvers with multi-agent reinforcement learning. In *FM*, volume 13047 of *Lecture Notes in Computer Science*, pages 103–121. Springer, 2021.
31. H. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the pentium processor. 1994.
32. M. Temel. MultGen. https://github.com/temelmertcan/multgen, 2020. SHA 32f4eb1bff419a88f02035d365d88089f6c33e5f.
33. A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC*, ESEC/FSE-7, page 253–267, Berlin, Heidelberg, 1999. Springer-Verlag.
34. A. Zeller. *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2021.
35. A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.
36. A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

Appendix A

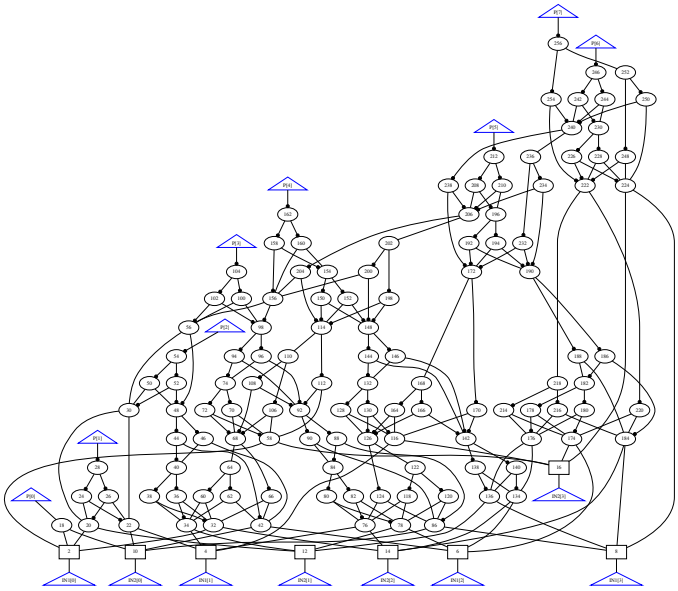


Fig. 4. AIG of "sp-ar-rc-4" multiplier used in the experiment of Tbl. 3

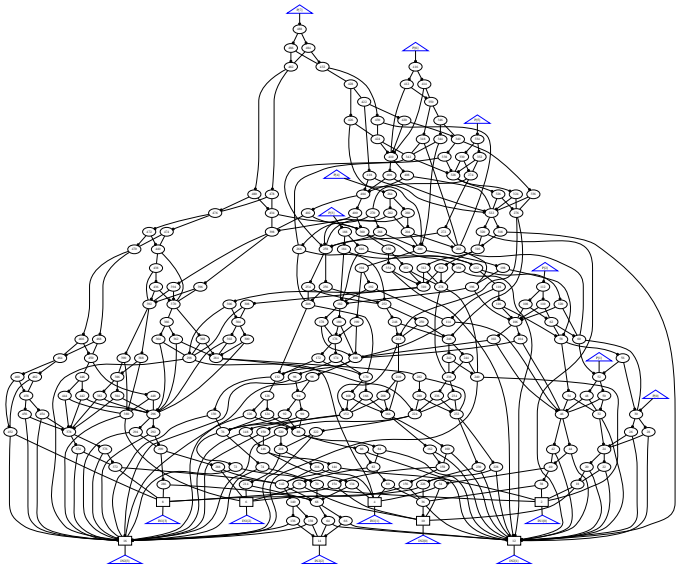
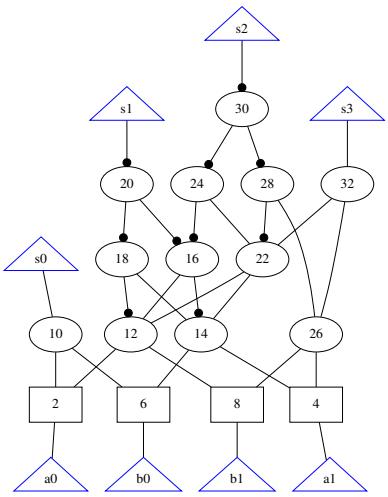


Fig. 5. AIG of "bp-ba-lf-4" multiplier used in the experiment of Tbl. 4

Appendix B




```
  AIGER ASCII
aag 16 4 0 4 12
2
4
6
8
10
21
31
32
10 6 2
12 8 2
14 6 4
16 15 12
18 14 13
20 19 17
22 14 12
24 22 17
26 8 4
28 26 23
30 29 25
32 26 22
i0 a0
i1 a1
i2 b0
i3 b1
o0 s0
o1 s1
o2 s2
o3 s3
```

```
        Simulation
======================
 a0 =  12  = 1     // input
 a1 =  14  = 1     // input
 b0 =  16  = 1     // input
 b1 =  18  = 1     // input
 s0 =  110 = 1
 s1 = ¬120 = 0
 s2 = ¬130 = 1
 s3 =  132 = 1
110 =  16 ∧ 12  = 1
112 =  18 ∧ 12  = 1
114 =  16 ∧ 14  = 1
116 = ¬114∧ 112 = 0
118 =  114∧¬112 = 0
120 = ¬118∧¬116 = 1
122 =  114∧ 112 = 1
124 =  122∧¬116 = 1
126 =  18 ∧ 14  = 1
128 =  126∧¬122 = 0
130 = ¬128∧¬124 = 0
132 =  126∧ 122 = 1
----------------------
Simulation result:
11 * 11 = 1101   // binary
 3 *  3 = 13     // decimal
```

Fig. 6. Incorrect multiplier which DyPoSub claims to be correct: AIG graph (left), AIGER ASCII encoding (middle), manual simulation (right)

Appendix C

Table 8. Multiple mutations for 6-bit multipliers

		AMulet2.1 [10,13]							AMulet2.2						DyPoSub [21]					
		✔		✗					✔		✗				✔		✗			
#mut	#	👍	👎	👍	👎	≮	↯	⧗	👍	👎	👍	👎	≮	↯	⧗	👍	👎	👍	👎	↯	⧗
2	205	2	83	0	0	10	57	53	2	174	0	0	0	0	29	2	199	0	0	0	4
4	197	1	38	0	0	7	104	47	1	132	0	0	0	0	64	1	190	0	0	0	6
6	166	0	15	0	0	7	97	47	0	100	0	0	0	0	66	0	159	0	0	0	7
8	218	0	7	0	0	4	158	49	0	97	0	0	0	0	121	0	194	0	0	0	24
10	200	0	4	0	0	5	158	33	0	82	0	0	0	0	118	0	177	0	0	0	23
12	213	0	6	0	0	3	171	33	0	64	0	0	0	0	149	0	177	0	0	0	36
14	195	0	2	0	0	0	162	31	0	52	0	0	0	0	143	0	156	0	0	1	38
16	208	0	0	0	0	1	183	24	0	52	0	0	0	0	156	0	164	0	0	1	43
18	196	0	1	0	0	0	178	17	0	33	0	0	0	0	163	0	150	0	0	0	46
20	202	0	0	0	0	1	190	11	0	35	0	0	0	0	167	0	135	0	0	0	67
2 000		3	156	0	0	38	1 458	345	3	821	0	0	0	0	1 176	3	1 701	0	0	2	294