Fuzzing and Delta Debugging And-Inverter Graph Verification Tools

Daniela Kaufmann[1] and Armin Biere[2]

[1] Johannes Kepler University Linz, Austria &
Software Competence Center Hagenberg, Austria
[2] Albert-Ludwigs-University Freiburg, Germany

Tests and Proofs
05. July 2022

Ensuring correctness of verification tools is
***equally** important as the correctness of the actual problems*
they try to establish.

Contributions

scch { }

Incentive:

- Incentive towards investing effort in automated testing and debugging of automated reasoning tools
- Focus is on hardware verification

Tools:

- Generation-based fuzzer *MultAIGenFuzzer*
- Mutation-based fuzzer *AIGoFuzzing*
- Delta debugging tool *AIGdd2*

Experiments:

- Evaluate presented fuzzing tools on multiplier verification tools
- Investigate correctness and robustness.

Correctness & Robustness

scch { }

Correct

- Tool returns “correct” for correct multipliers.
- Tool returns “incorrect” for incorrect multipliers.

Robust

- Tool does not crash on inputs.

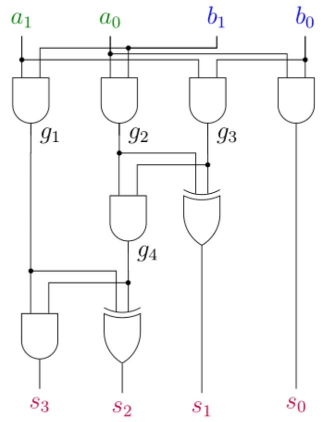
Use Case: Multiplier Verification

scch { }

Given: Gate-level multiplier C for fixed bit-width.

Question: For all possible $a_i, b_i \in \mathbb{B}$:

$$(2a_1 + a_0) * (2b_1 + b_0) = 8s_3 + 4s_2 + 2s_1 + s_0?$$



- In recent years verification of gate-level integer multipliers has made significant progress:
  - [SayedGroßeKühneSoekenDrechsler-DATE16], [SayedGroßenSoekenDrechsler-FMCAD16], [RitircBiereKauers-FMCAD17], [MahzoonGroßeDrechsler-ICCAD18], [RitircBiereKauers-DATE18], [MahzoonGroßeDrechsler-DAC19], [KaufmannBiereKauers-FMCAD19], [MahzoonGroßeSchollDrechsler-DATE20], [KaufmannBiere-TACAS21], [MahzoonGroßeDrechsler-TCAD21], [KaufmannBeameBiereNordström-DATE22]

  - [MahzoonGroßeSchollDrechsler-DATE20] → **DyPoSub**
  - [KaufmannBiere-TACAS21] → **AMulet2**

Related Work

scch { }

Automated Reasoning & Fuzzing

- Satisfiability Modulo Theories (SMT) [BrummayerBiere-SMTWorkshop09], [MansurChristakisWüstholzZhang-FSE2020]
- Satisfiability Checking (SAT) [BrummayerLonsingBiere-SAT10]
- Quantified Boolean formulas (QBF) [BrummayerLonsingBiere-SAT10]
- Interactive Provers [LampropoulosHicksPierce-OOPSLA19]

- Current research in multiplier verification focuses on efficiency and automation.
- We are not aware of research of fuzzing and debugging for tools that read AIGs.

scch { }

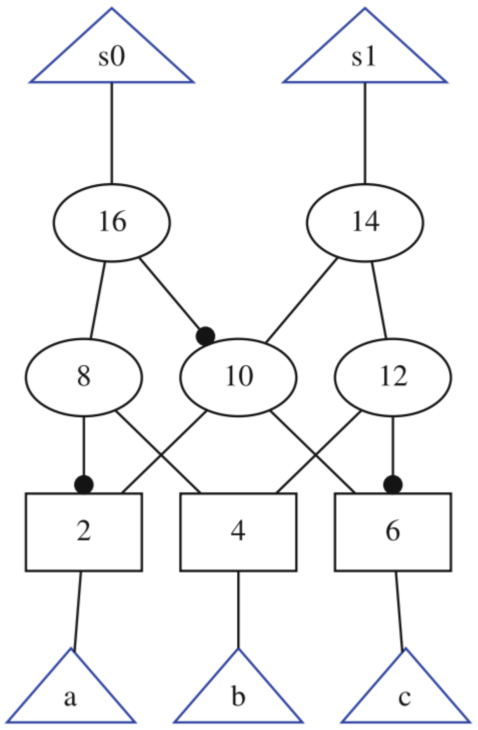
AIGs, Fuzzing & Delta Debugging

And-Inverter Graphs - AIGs

scch { }

- [KuehlmannParuthiKrohmGanai-TCAD02]
- Directed acyclic graph
- Represents the structural implementation of a circuit
- Rarely structural efficient, but efficient to manipulate

- Consists of two-input nodes
- Nodes represent logical conjunction $l_{14} = l_{10} \wedge l_{12}$
- Markings on edges represent negation $l_{16} = l_8 \wedge \neg l_{10}$



Fuzzing

scch { }

- Technique for automated software testing

- **Idea:**
  - Treat the program as a black-box
  - Use random, invalid and unexpected inputs
  - Detect failures and tool crashes

- **History:**
  - Originated in the 90’s: random inputs detected many errors in UNIX command line programs
  - Since then, a variety of automated testing approaches and tools have been developed (ClusterFuzz by Google, or OneFuzz by Microsoft)

Fuzzing Techniques

scch { }

Input Usage	Generation-based fuzzers	generate random input from scratch
	Mutation-based fuzzers	mutate existing input seeds by making small modifications
Structural Knowledge	Black-box fuzzers	are completely unaware of the internal structure of the program under test
	White-box fuzzers	uses program analysis to systematically generate inputs that increase code coverage

Black-box fuzzing is faster and can easily be parallelized;
but may only trigger easy-to-reach bugs.

Fuzzing Tools

MultAIGenFuzzer

- Generation-based
- Black-box

Generates multiplier circuits from scratch by combining building blocks.

AIGoFuzzing

- Mutation-based
- Black-box

Mutates AIGs without violating structural constraints.

Delta Debugging

scch { }

- Aims to reduce manual workload of debugging software problems
- Minimizes failure-inducing inputs

- **Idea:**
  - Binary search strategy
  - Repeatedly remove smaller and smaller parts of the failure inducing input
  - Until a minimal fix point is reached.

- AIGdd2 removes AIG nodes to narrow down the failure cause.

Tools

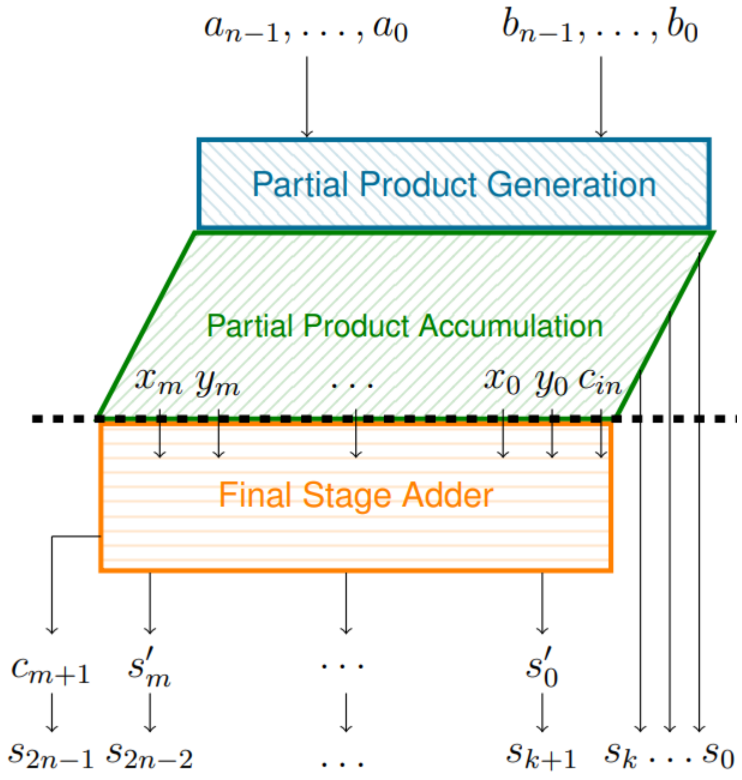
scch { }

MultAIGenFuzzer, AIGoFuzzing, AIGdd2

Multipliers

scch { }

$$(\sum_{i=0}^{n-1} 2^i a_i) * (\sum_{i=0}^{n-1} 2^i b_i) = \sum_{i=0}^{2n-1} 2^i s_i$$

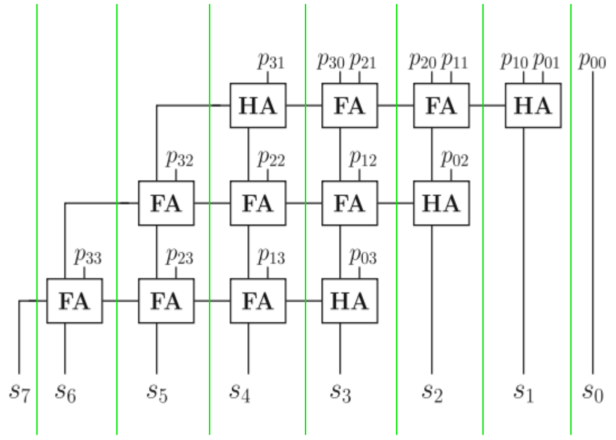


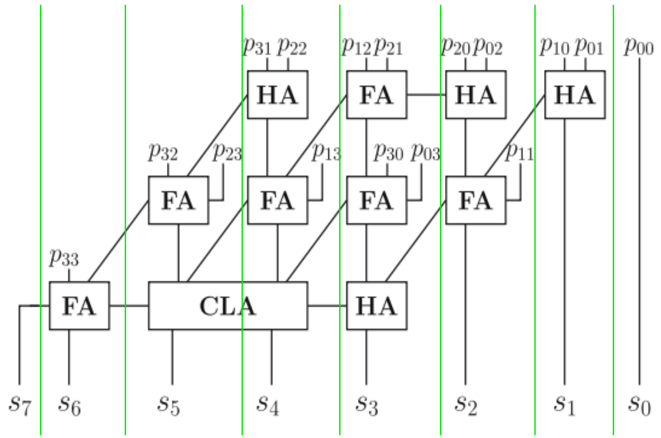
- For each component, several algorithms are available
  - Partial Product Generation: Simple conjunction, Radix Booth Encoding
  - Partial Product Accumulation: Array, Wallace-tree, compressor trees, …
  - Final Stage Adder: Ripple-Carry, Carry-lookahead, Ladner-Fischer,…
- All components have certain patterns
  - Their number is limited
  - Danger of introducing a bias in verification algorithms

Generation-based Fuzzer - MultAIGenFuzzer

- Generate correct multipliers with random patterns

- Random multiplier generation using MultAIGenFuzzer:
  - Partial product generation:
    - Generate partial products $p_{ij} = a_i b_j$ using simple conjunction
    - Assign partial products to slices
  - Partial product accumulation
    - Select two or three random elements of a random slice
    - Addition using half- and full-adders
    - Repeat this step until all slices contain at maximum two elements
  - Final Stage adder
    - Using a mixture of full-adders, half-adders and carry-lookahead adders





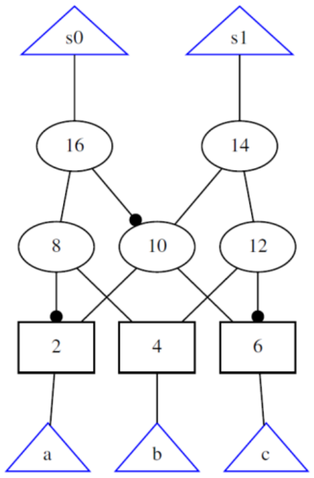
Mutation-based Fuzzing - AIGoFuzzing

scch { }

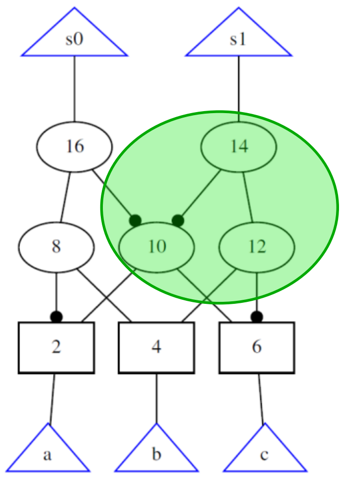
- Input: AIGs
- Small modifications in the AIG that may or may not change the specification
- Not specifically designed for multiplier verification and can be used on any given AIG

AIGoFuzzing - Mutations

scch { }



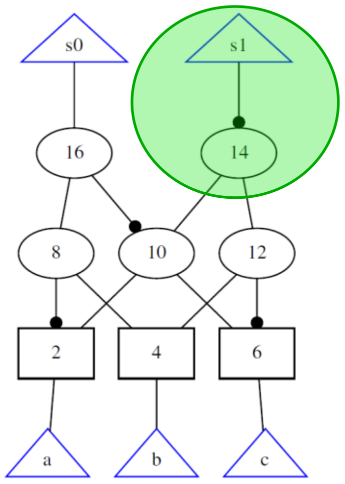
Seed



Swapping Signs

$$l_{14} = l_{10} \wedge l_{12}$$
$$\Rightarrow$$
$$l_{14} = \neg l_{10} \wedge l_{12}$$

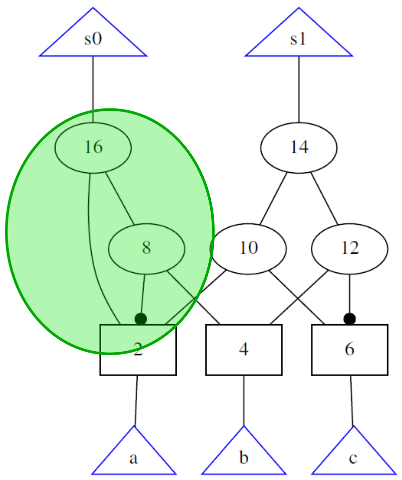
Affects specification



Swapping Output

- Special case of Swapping signs
- Negates specification for AIGs with single outputs

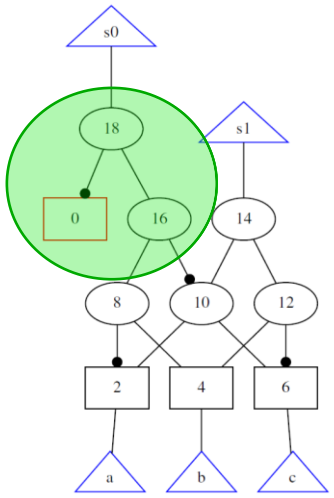
Affects specification



Modifying Node Input

$$l_{16} = l_8 \wedge \neg l_{10}$$
$$\Rightarrow$$
$$l_{16} = l_8 \wedge l_2$$

Affects specification



Inserting a constant

- Four cases: ({0,1},{∧,∨})
- $v = v \vee 0 = v \wedge 1$
- $v \neq v \vee 1 \neq v \wedge 0$

Affects specification in 50% of the cases

Delta Debugging with Slices – AIGdd2

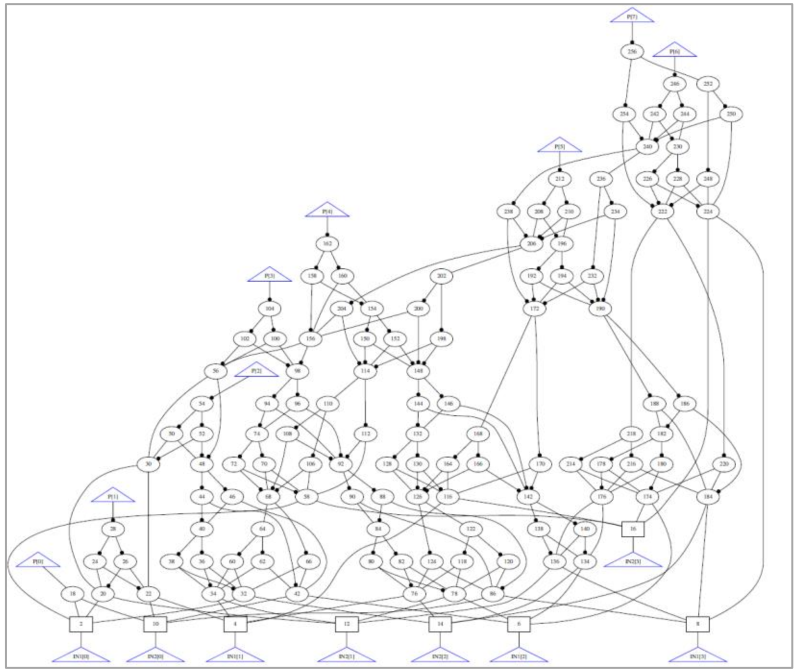
scch { }

- Minimizes failure-inducing AIGs while preserving errors
- AIGdd2 does not find THE smallest possible failure-inducing input, but it will find a minimal example.
- Re-implementation of AIGdd [BiereHeljankoWieringa-FMV11]

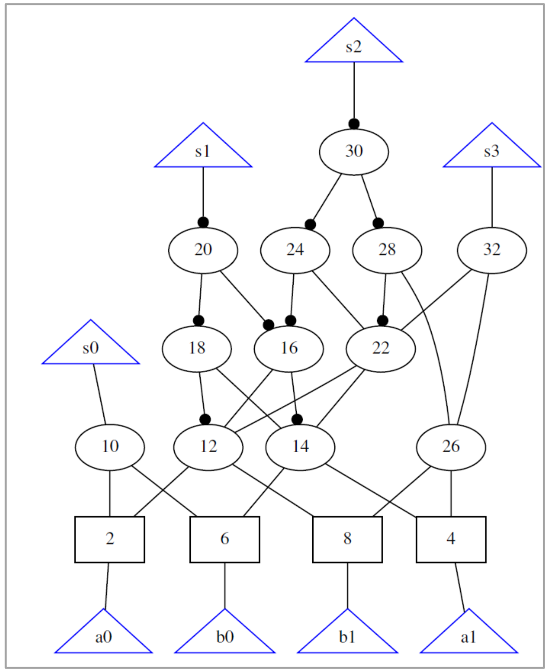
- Novel:
  - Option to limit structural changes of the AIG
  - Slicing based delta debugging approach that allows us to shrink the bit-width of multipliers
  - Set most significant output and input bits to 0 and propagate.

- Afterwards we use binary-search based approach of AIGdd to further shrink the size of the sliced AIG

Delta Debugging with Slices – AIGdd2



4-bit multiplier



2-bit multiplier

Fuzzing, Tests & Proofs

Reduction Engines

DyPoSub

Input
- Multiplier given as AIG

Verification
- Polynomial Encoding
- Reduction using **dynamic** ordering

Output
- Yes/No Answer

AMulet2

Input
- Multiplier given as AIG

Verification
- Polynomial Encoding
- Reduction using **static** ordering

Output
- Yes/No Answer
- Proof in case of Yes-Answer
- Counterexample in case of No-Answer

Fuzzing, Tests and Proofs

scch { }

Evaluate robustness and correctness of AMulet2 and DyPoSub

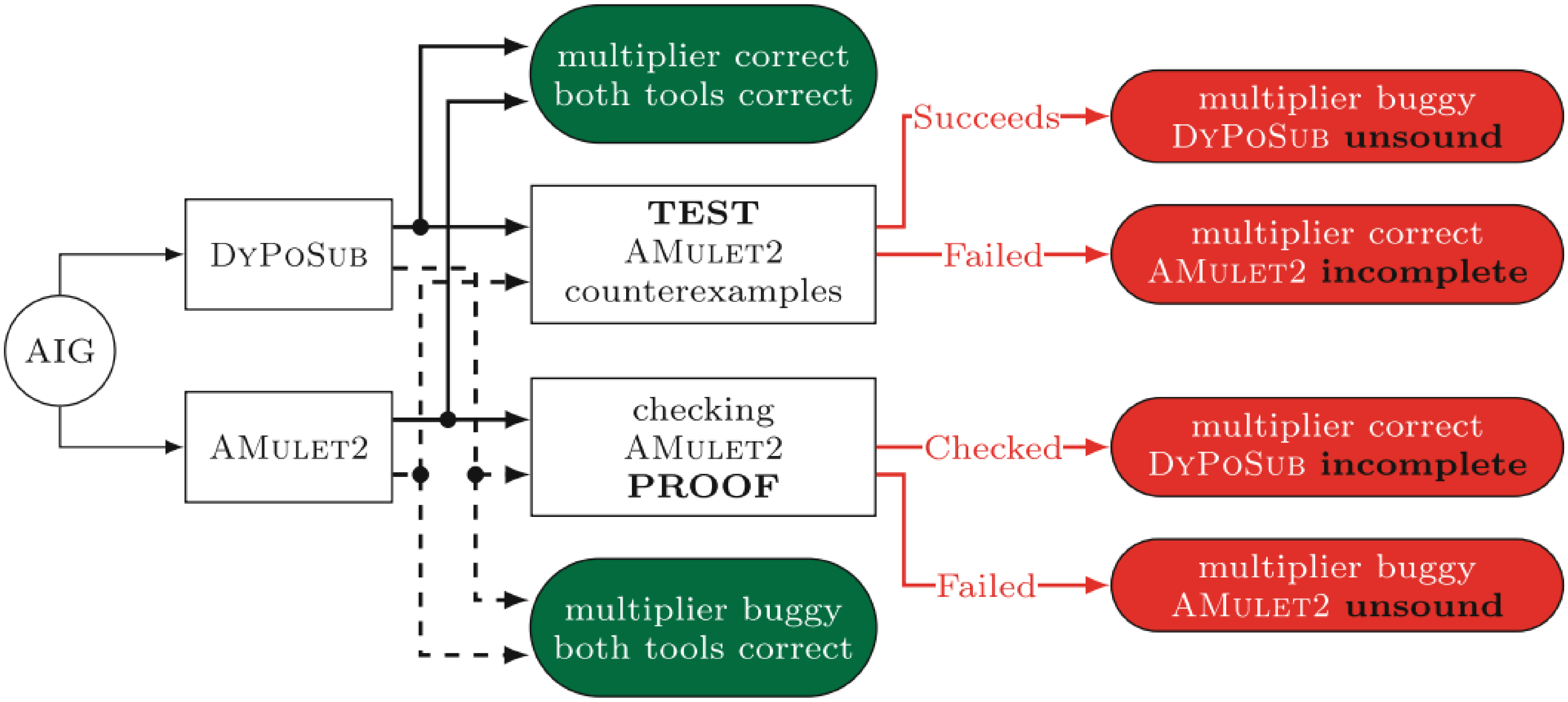
- Fix possible errors in AMulet2.1 and release AMulet2.2

1. Robustness:
   - Use MultAIGenFuzzer to detect overfitting of reduction algorithms
   - Use AIGoFuzzing to detect crashes
   - Combine AIGoFuzzing and MultAIGenFuzzer

2. Correctness:
   - We use differential testing to deduce the correctness of both tools.

Differential Testing

scch { }



Evaluation

Experiments

Table 1. MultAIGenFuzzer benchmarks without carry-lookahead modules.

		AMulet2.1 [10, 13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
n	#	👍	👎	🖒	🖓	≮	⚡	⌛	👍	👎	🖒	🖓	≮	⚡	⌛	👍	👎	🖒	🖓	⚡	⌛
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

- AMulet2.1 is robust and correct
- AMulet2.2 is robust and correct
- DyPoSub is robust and correct

✔	✗
👍 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

Experiments

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

- AMulet2.1 has time-outs → overfitting
- AMulet2.2 has time-outs → overfitting
- DyPoSub is robust and correct

✔	✗
👍 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)
	ϟ segmentation fault
	⧗ exceeding the time limit

Experiments

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

mutation	#	AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
		👍	👎	👍	👎	≮	⚡	⌛	👍	👎	👍	👎	≮	⚡	⌛	👍	👎	👍	👎	⚡	⌛
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

- AMulet2.1 is not robust
- AMulet2.2 is robust and correct
- DyPoSub is unsound on 6 benchmarks

✔	✗
👍 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

Experiments

scch { }

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

Mutation	#	AMulet2.1 [10,13]							AMulet2.2							DyPoSub [21]					
		✔		✗					✔		✗					✔		✗			
		👍	👎	👍	👎	≮	⚡	⌛	👍	👎	👍	👎	≮	⚡	⌛	👍	👎	👍	👎	⚡	⌛
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

- AMulet2.1 is not robust
- AMulet2.2 is robust and correct
- DyPoSub is incomplete on 258 benchmarks

✔	✗
👍 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

Experiments – Delta Debugging

scch { }

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

		-slicing						+slicing					
	mutation	⌛	✗	✔				⌛	✗	✔			
				#	min(%)	max(%)	avg(%)			#	min(%)	max(%)	avg(%)
Table 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(%)
Table 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(%)
Table 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	-	-	-

Summary of the Experiments

Generation-based Fuzzing

- AMulet2 is overfitted to existing FSAs.
- DyPoSub is robust and correct on these benchmarks

Mutation-based Fuzzing

- AMulet2.1 has robustness issues, which could be fixed in AMulet2.2
- DyPoSub is unsound and incomplete

Conclusion

- Software is only as good as its robustness and correctness
- Generation- and mutation-based fuzzing techniques randomly generate input to tackle issues
- Delta debugging allows us to generate smaller failure-inducing benchmarks

Observation:

- Randomly shuffling the structure of available inputs helps to avoid overfitting
- Even small mutations can reveal defects efficiently
- Verification tools need to produce proof certificates to prevent false results
- Shrinking failure-inducing inputs using delta debugging allows to zoom in on defects