Improved Test Pattern Generation for Hardware Trojan Detection using Genetic Algorithm and Boolean Satisfiability

Sayandeep Saha, Rajat Subhra Chakraborty
Srinivasa Shashank Nuthakki, Anshul
and Debdeep Mukhopadhyay

Secure Embedded Architecture Laboratory (SEAL)
Indian Institute of Technology, Kharagpur
Kharagpur, India

September 16, 2015

Outline

- Introduction
- Motivation
- Logic Testing Based Trojan Detection
- Scopes of Improvement
- Proposed New Strategy
- Experimental Results
- Conclusion

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].
  - Stealthy in nature.

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].
  - Stealthy in nature.
  - Bypass conventional design verification and post-manufacturing tests.

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].
  - Stealthy in nature.
  - Bypass conventional design verification and post-manufacturing tests.
  - Effect:

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].
  - Stealthy in nature.
  - Bypass conventional design verification and post-manufacturing tests.
  - Effect:
    - **Functional failure**

Introduction: Hardware Trojan Horse

- Modern Semiconductor industry trends:
  - Outsourcing of the Fabrication facility.
  - Procurement of third party intellectual property (3PIP) cores.
- **Threats**: Malicious tampering called **Hardware Trojan Horses** (HTH) [1].
  - Stealthy in nature.
  - Bypass conventional design verification and post-manufacturing tests.
  - Effect:
    - **Functional failure**
    - **Leakage of secret information**

Motivation

- **Side-channel techniques**:

Motivation

- **Side-channel techniques**:
  - Most widely explored.

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]
- **Logic testing based techniques**:

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]
- **Logic testing based techniques**:
  - Does not need design modification.

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]
- **Logic testing based techniques**:
  - Does not need design modification.
  - Only means of detecting extremely small Trojans even with 1-2 gates [5].

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]
- **Logic testing based techniques**:
  - Does not need design modification.
  - Only means of detecting extremely small Trojans even with 1-2 gates [5].
  - May be used to amplify the effectiveness of side-channel tests [5].

Motivation

- **Side-channel techniques**:
  - Most widely explored.
  - Not suitable for extremely small Trojans [2].
- **DFT techniques**:
  - For run-time/test-time detection and/or prevention.
  - Suffers from security threats from Trojans itself [3, 4]
- **Logic testing based techniques**:
  - Does not need design modification.
  - Only means of detecting extremely small Trojans even with 1-2 gates [5].
  - May be used to amplify the effectiveness of side-channel tests [5].
- **Surprisingly, very few works has been done on Logic testing based Trojan detection**.

Logic Testing Based Trojan Detection: Problem Statement

- Generate tests to trigger a Trojan and observe its effect at the output.

Logic Testing Based Trojan Detection: Problem Statement

- Generate tests to trigger a Trojan and observe its effect at the output.
- Trojans are triggered by extremely rare logic events inside the circuit:

Logic Testing Based Trojan Detection: Problem Statement

- Generate tests to trigger a Trojan and observe its effect at the output.
- Trojans are triggered by extremely rare logic events inside the circuit:
  - Can be achieved by activating some of the low transition nets simultaneously to there rare logic values (**Simultaneous activation of rare logic conditions (rare nodes)**).

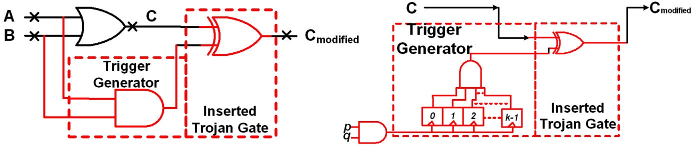
Logic Testing Based Trojan Detection: Problem Statement

- Generate tests to trigger a Trojan and observe its effect at the output.
- Trojans are triggered by extremely rare logic events inside the circuit:
  - Can be achieved by activating some of the low transition nets simultaneously to there rare logic values (**Simultaneous activation of rare logic conditions (rare nodes)**).
- Number of such possible triggers are exponential in the number of low transition nets.

Logic Testing Based Trojan Detection: Problem Statement

- Generate tests to trigger a Trojan and observe its effect at the output.
- Trojans are triggered by extremely rare logic events inside the circuit:
  - Can be achieved by activating some of the low transition nets simultaneously to there rare logic values (**Simultaneous activation of rare logic conditions (rare nodes)**).
- Number of such possible triggers are exponential in the number of low transition nets.
- A candidate trigger may or may not constitute a feasible trigger.

Logic Testing Based Trojan Detection: Trojan Models

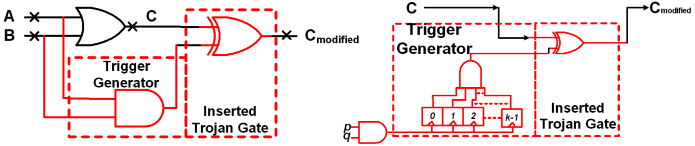


(a) Combinational Trojan

(b) Sequential (counter-based) Trojan

- **Trigger inputs A and B**: internal rare nodes inside the circuit.

Logic Testing Based Trojan Detection: Trojan Models



(a) Combinational Trojan (b) Sequential (counter-based) Trojan

- **Trigger inputs A and B**: internal rare nodes inside the circuit.
- **Sequential Trojan**: activated if rare logic condition occurs *k* times.

Logic Testing Based Trojan Detection: Previous Works

- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].

Logic Testing Based Trojan Detection: Previous Works

- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].
- Utilized: **Simultaneous activation of rare nodes for triggering**.

Logic Testing Based Trojan Detection: Previous Works

- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].
- Utilized: **Simultaneous activation of rare nodes for triggering**.
- Rare nodes are selected based on a **“rareness threshold”** (θ).

Logic Testing Based Trojan Detection: Previous Works

- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].
- Utilized: **Simultaneous activation of rare nodes for triggering**.
- Rare nodes are selected based on a **"rareness threshold"** (θ).
- ***N-detect*** ATPG scheme was proposed:

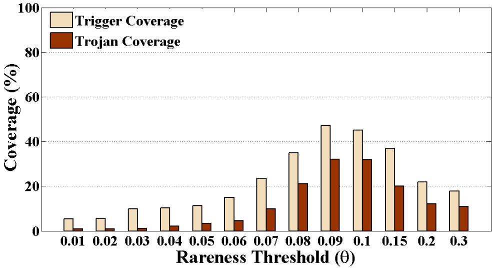
Logic Testing Based Trojan Detection: Previous Works

- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].
- Utilized: **Simultaneous activation of rare nodes for triggering**.
- Rare nodes are selected based on a **"rareness threshold"** (θ).
- ***N-detect*** ATPG scheme was proposed:
  - To individually activate a set of rare nodes to their rare values at least N-times.

Logic Testing Based Trojan Detection: Previous Works

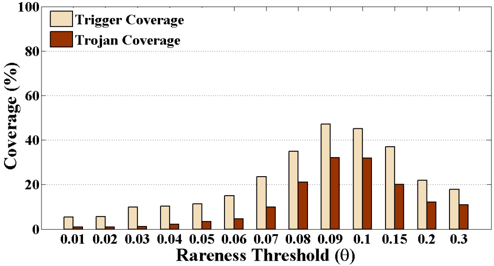
- Chakraborty et.al presented an automatic test pattern generation (ATPG) scheme called *MERO* (*CHES* 2009) [5].
- Utilized: **Simultaneous activation of rare nodes for triggering**.
- Rare nodes are selected based on a **"rareness threshold"** (θ).
- ***N-detect*** ATPG scheme was proposed:
  - To individually activate a set of rare nodes to their rare values at least N-times.
- **Assumption**: Multiple individual activation also increases the probability of simultaneous activation.

Scopes of Improvement



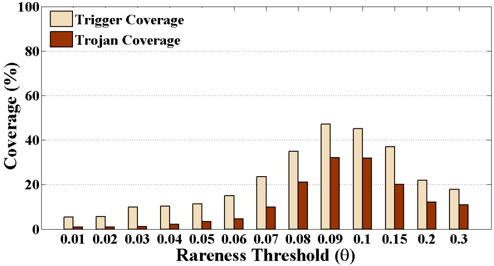
- **Trojan test set**: only “hard-to-trigger” Trojans with triggering probability (P_{tr}) below 10^{-6}.

Scopes of Improvement



- **Trojan test set**: only "hard-to-trigger" Trojans with triggering probability (P_{tr}) below 10^{-6}.
- Best coverage achieved near $\theta = 0.1$ for most of the circuits– **best operating point**.

Scopes of Improvement



- **Trojan test set**: only "hard-to-trigger" Trojans with triggering probability (P_{tr}) below 10^{-6}.
- Best coverage achieved near $\theta = 0.1$ for most of the circuits– **best operating point**.
- Test Coverage of *MERO* is consistently below 50% for circuit c7552.

Proposed Solutions

Proposed Solutions

- Simultaneous activation of rare nodes: **in a direct manner.**

Proposed Solutions

- Simultaneous activation of rare nodes: **in a direct manner.**

- Replacement of the *MERO* heuristics with a combined Genetic algorithm (GA) and boolean satisfiability (SAT) based scheme.

Proposed Solutions

- Simultaneous activation of rare nodes: **in a direct manner**.

- Replacement of the *MERO* heuristics with a combined Genetic algorithm (GA) and boolean satisfiability (SAT) based scheme.

- Refinement of the test set considering the “payload effect” of Trojans: **a fault simulation based approach**.

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.
  - Does not guarantee the detection of all possible faults, especially for those which are hard to detect.

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.
  - Does not guarantee the detection of all possible faults, especially for those which are hard to detect.
- **SAT based test generation**:

Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.
  - Does not guarantee the detection of all possible faults, especially for those which are hard to detect.
- **SAT based test generation**:
  - Remarkably useful for hard-to-detect faults.

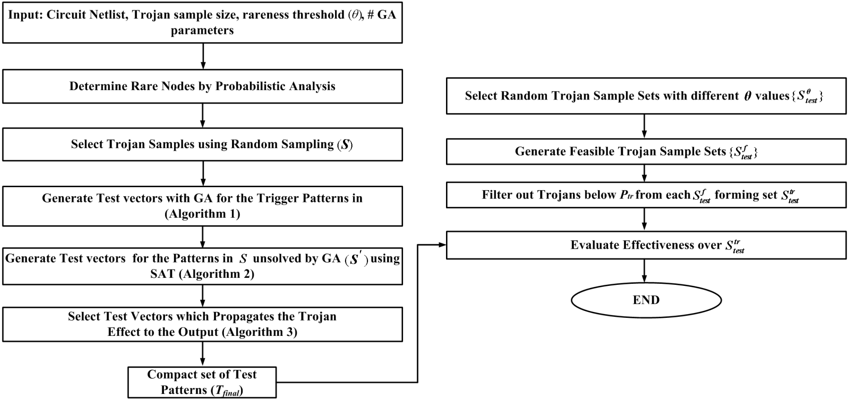
Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.
  - Does not guarantee the detection of all possible faults, especially for those which are hard to detect.
- **SAT based test generation**:
  - Remarkably useful for hard-to-detect faults.
  - Targets the faults one by one– **incurs higher execution time for large fault lists**.

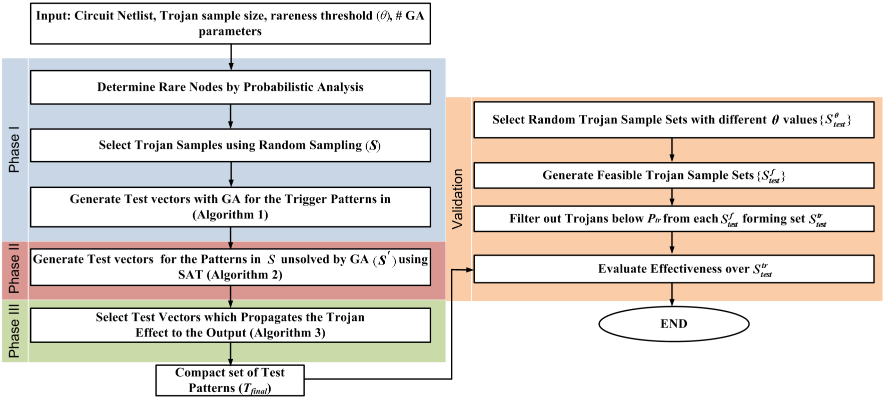
Genetic Algorithm and Boolean Satisfiability for ATPG

- **GA in ATPG**:
  - Achieves reasonably good test coverage over the fault list very quickly.
  - Inherently parallel, and rapidly explores search space.
  - Does not guarantee the detection of all possible faults, especially for those which are hard to detect.
- **SAT based test generation**:
  - Remarkably useful for hard-to-detect faults.
  - Targets the faults one by one– **incurs higher execution time for large fault lists**.
- We combine the “best of both worlds” for GA and SAT.

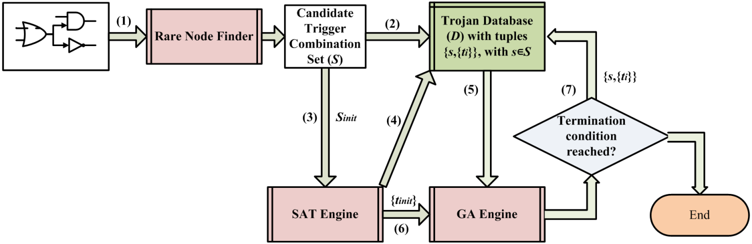
Proposed Scheme



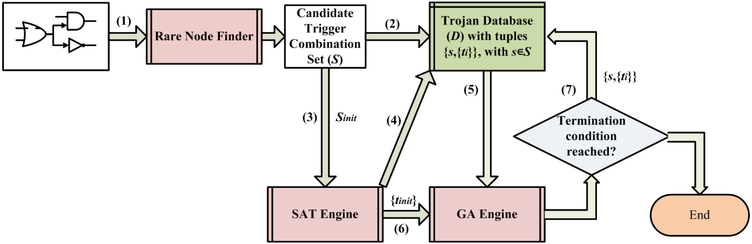
Proposed Scheme



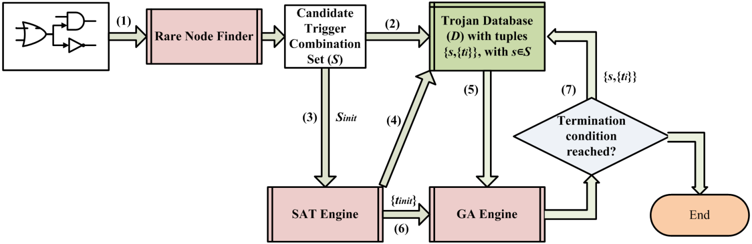
Phase I: Genetic Algorithm



Phase I: Genetic Algorithm

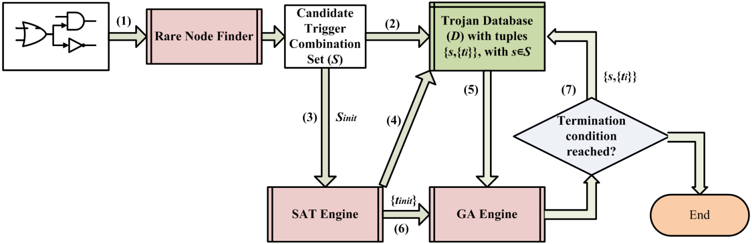


Phase I: Genetic Algorithm



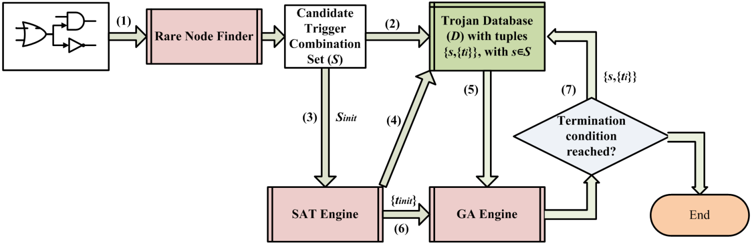
- Rare nodes are found using a probabilistic analysis as described in [6].

Phase I: Genetic Algorithm



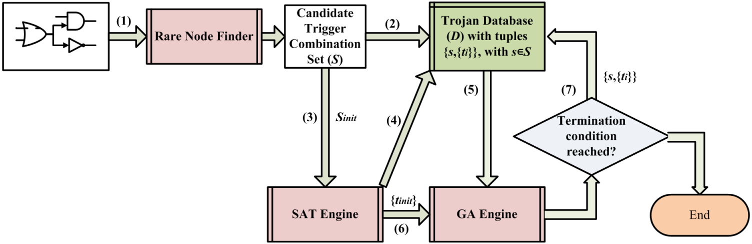
- Rare nodes are found using a probabilistic analysis as described in [6].
- GA dynamically updates the database with test vectors for each trigger combination.

Phase I: Genetic Algorithm



- Rare nodes are found using a probabilistic analysis as described in [6].
- GA dynamically updates the database with test vectors for each trigger combination.

Phase I: Genetic Algorithm



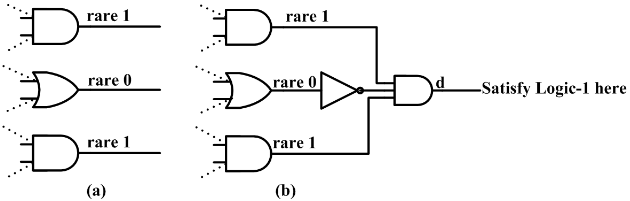
- Rare nodes are found using a probabilistic analysis as described in [6].
- GA dynamically updates the database with test vectors for each trigger combination.
- **Termination**: if either 1000 generations has been reached or a specified $\#T$ number of test vectors has been generated.

Phase I: Genetic Algorithm

How a SAT Instance is Formed?

Phase I: Genetic Algorithm

How a SAT Instance is Formed?



Phase I: Genetic Algorithm

Phase I: Genetic Algorithm

Goal 1

- An effort to generate test vectors that would activate the most number of sampled trigger combinations.

Phase I: Genetic Algorithm

Goal 1

- An effort to generate test vectors that would activate the most number of sampled trigger combinations.

Goal 2

- An effort to generate test vectors for hard-to-trigger combinations.

Phase I: Genetic Algorithm

Phase I: Genetic Algorithm

Fitness Function

$$f(t) = R_{count}(t) + w * I(t) \quad (1)$$

- $f(t)$: fitness value of a test vector t.
- $R_{count}(t)$: the number of rare nodes triggered by the test vector t.
- w : constant scaling factor (> 1).
- $I(t)$: *relative improvement* of the database $\mathcal{D}$ due to the test vector t.

Phase I: Genetic Algorithm

Phase I: Genetic Algorithm

Relative Improvement

$$I(t) = \frac{n_2(s) - n_1(s)}{n_2(s)} \tag{2}$$

- $n_1(s)$: number of test patterns in bin s before update
- $n_2(s)$: number of test patterns in bin s after update.

Phase I: Genetic Algorithm

Crossover and Mutation

Phase I: Genetic Algorithm

Crossover and Mutation

- Two-point binary crossover with probability 0.9.

Phase I: Genetic Algorithm

Crossover and Mutation

- Two-point binary crossover with probability 0.9.
- Binary mutation with probability 0.05.

Phase I: Genetic Algorithm

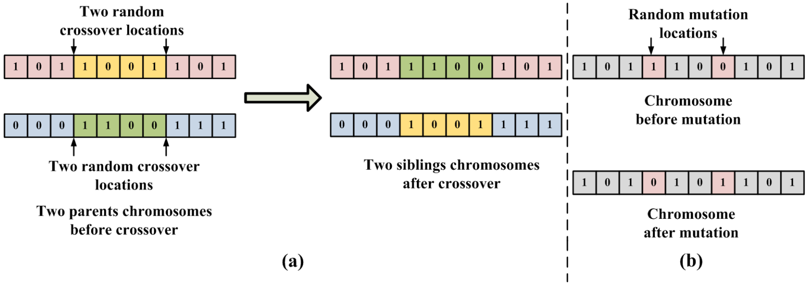
Crossover and Mutation

- Two-point binary crossover with probability 0.9.
- Binary mutation with probability 0.05.
- Population size: 200 (combinatorial), 500 (sequential).

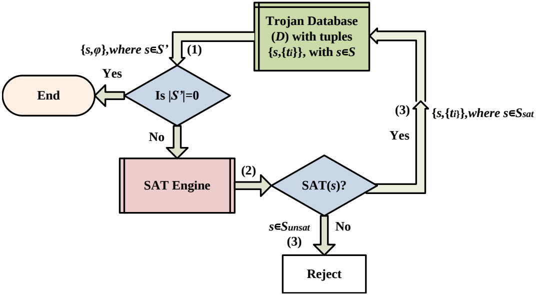
Phase I: Genetic Algorithm

Crossover and Mutation

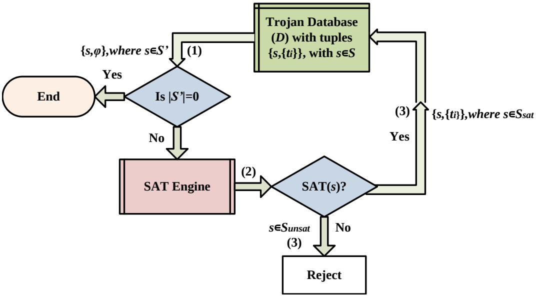
- Two-point binary crossover with probability 0.9.
- Binary mutation with probability 0.05.
- Population size: 200 (combinatorial), 500 (sequential).



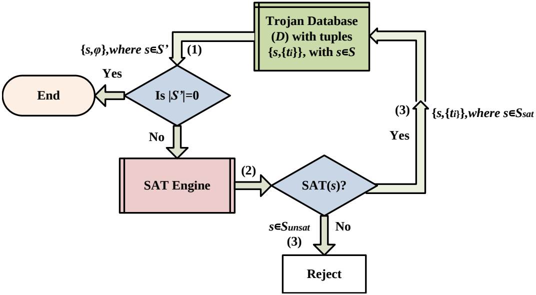
Phase II: Solving "Hard-to-Trigger" Patterns using SAT



Phase II: Solving "Hard-to-Trigger" Patterns using SAT

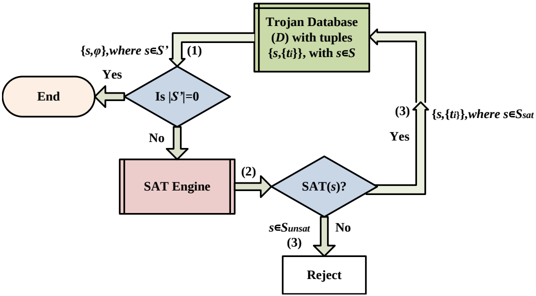


Phase II: Solving “Hard-to-Trigger” Patterns using SAT



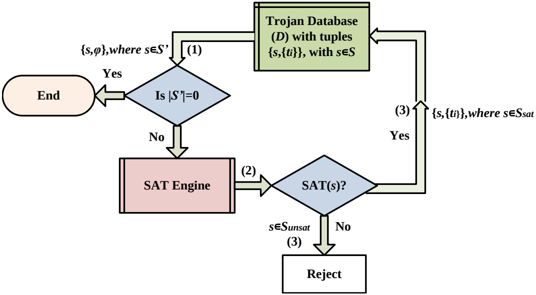
- $\mathcal{S}' \subseteq \mathcal{S}$ denotes the set of trigger combinations unresolved by GA.

Phase II: Solving "Hard-to-Trigger" Patterns using SAT



- $\mathcal{S}' \subseteq \mathcal{S}$ denotes the set of trigger combinations unresolved by GA.
- $\mathcal{S}_{sat} \subseteq \mathcal{S}'$ is the set solved by SAT.

Phase II: Solving "Hard-to-Trigger" Patterns using SAT



- $\mathcal{S}' \subseteq \mathcal{S}$ denotes the set of trigger combinations unresolved by GA.
- $\mathcal{S}_{sat} \subseteq \mathcal{S}'$ is the set solved by SAT.
- $\mathcal{S}_{unsat} \subseteq \mathcal{S}'$ remains unsolved and gets rejected.

Phase III: Payload Aware Test Vector Selection

Phase III: Payload Aware Test Vector Selection

- For a node to be payload:

Phase III: Payload Aware Test Vector Selection

- For a node to be payload:
  - **Necessary condition**: topological rank must be higher than the topologically highest node of the trigger combination.

Phase III: Payload Aware Test Vector Selection

- For a node to be payload:
  - **Necessary condition**: topological rank must be higher than the topologically highest node of the trigger combination.
- Not a sufficient condition.

Phase III: Payload Aware Test Vector Selection

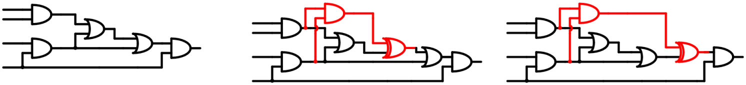
- For a node to be payload:
  - **Necessary condition**: topological rank must be higher than the topologically highest node of the trigger combination.
- Not a sufficient condition.

- In general, a successful Trojan triggering event provides no guarantee regarding its propagation to the primary output to cause functional failure of the circuit.

Phase III: Payload Aware Test Vector Selection

Phase III: Payload Aware Test Vector Selection

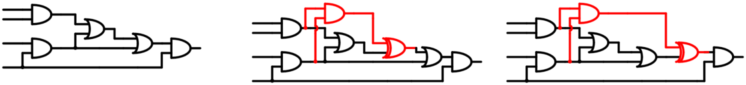
An Example



(a) (b) (c)

Phase III: Payload Aware Test Vector Selection

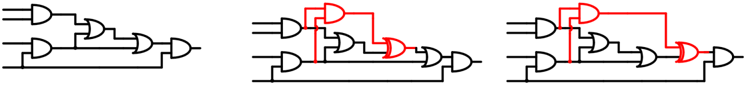
An Example



(a) (b) (c)

Phase III: Payload Aware Test Vector Selection

An Example

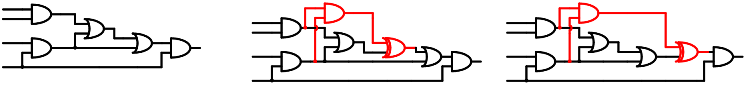


(a) (b) (c)

- Trojan is triggered by an input vector 1111.

Phase III: Payload Aware Test Vector Selection

An Example

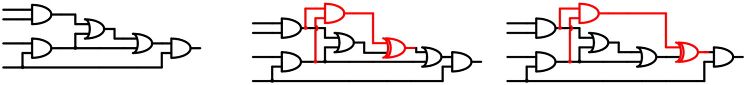


(a) (b) (c)

- Trojan is triggered by an input vector 1111.
- Payload-1 (Fig. (b)) has no effect on the output.

Phase III: Payload Aware Test Vector Selection

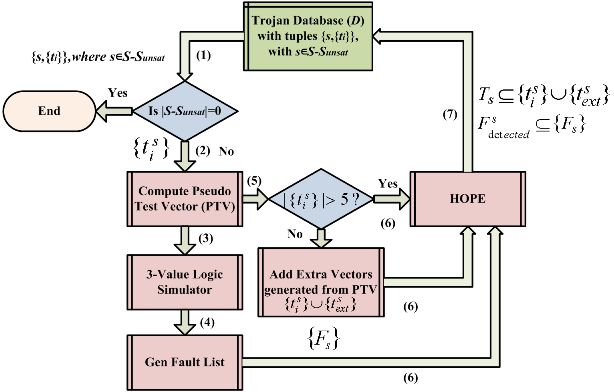
An Example



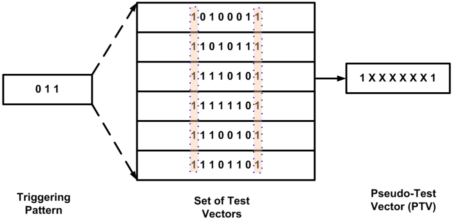
(a) (b) (c)

- Trojan is triggered by an input vector 1111.
- Payload-1 (Fig. (b)) has no effect on the output.
- Payload-2 (Fig. (c)) affects the output.

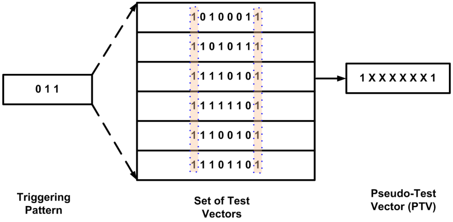
Phase III: Payload Aware Test Vector Selection



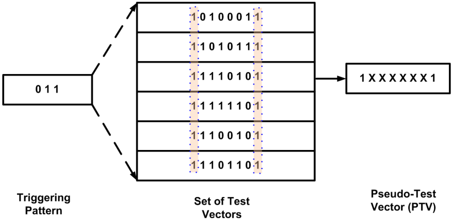
Phase III: Pseudo Test Vector



Phase III: Pseudo Test Vector

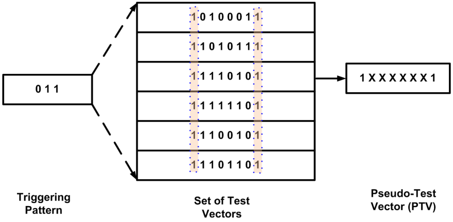


Phase III: Pseudo Test Vector



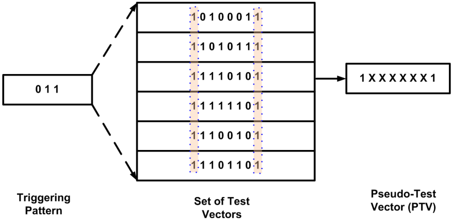
- For each set of test vectors ($\{t_i^s\}$) corresponding to a triggering combination (s), we find out the primary input positions which remains static (logic-0 or logic-1).

Phase III: Pseudo Test Vector



- For each set of test vectors ($\{t_i^s\}$) corresponding to a triggering combination (s), we find out the primary input positions which remains static (logic-0 or logic-1).
- Rest of the input positions are marked as "don't care" (X).

Phase III: Pseudo Test Vector



- For each set of test vectors ($\{t_i^s\}$) corresponding to a triggering combination (s), we find out the primary input positions which remains static (logic-0 or logic-1).
- Rest of the input positions are marked as "don't care" (X).
- A 3-value logic simulation is performed with this PTV and values of all internal nodes are noted down (0,1, or X).

Phase III: Payload Aware Test Vector Selection

The Fault list $\mathcal{F}_s$

Phase III: Payload Aware Test Vector Selection

The Fault list $\mathcal{F}_s$

- If the value at that node is 1, consider a stuck-at-zero fault there.

Phase III: Payload Aware Test Vector Selection

The Fault list $\mathcal{F}_s$

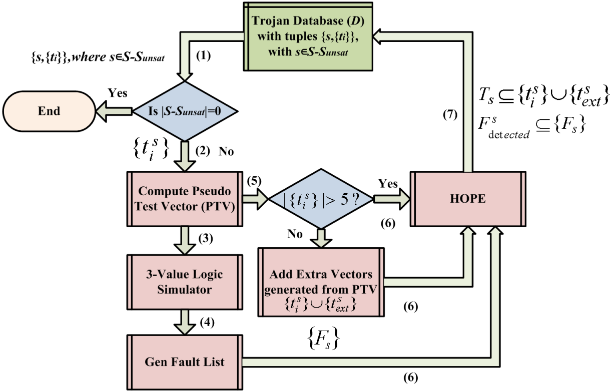
- If the value at that node is 1, consider a stuck-at-zero fault there.
- If the value at that node is 0, consider a stuck-at-one fault there.

Phase III: Payload Aware Test Vector Selection

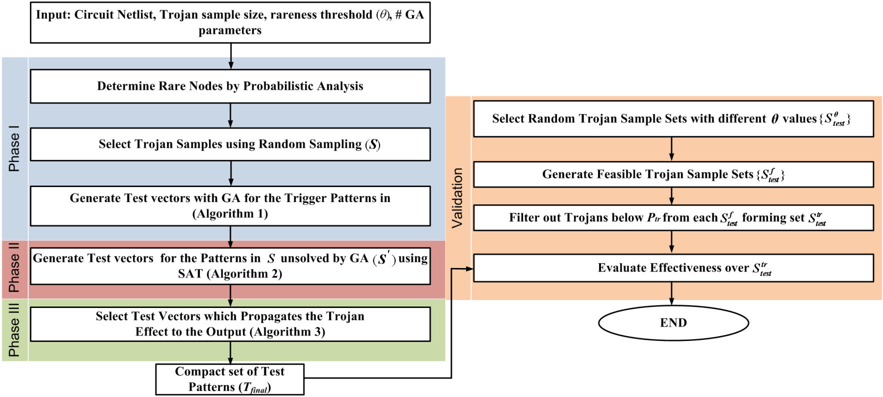
The Fault list $\mathcal{F}_s$

- If the value at that node is 1, consider a stuck-at-zero fault there.
- If the value at that node is 0, consider a stuck-at-one fault there.
- If the value at that node is X, consider a both stuck-at-one and stuck-at-zero fault at that location.

Phase III: Payload Aware Test Vector Selection



Experimental Results: Setup



Experimental Results: Setup

Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{\textit{test}}| = |\mathcal{S}| = 100000$ for each θ.

Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{\mathit{test}}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.

Experimental Results: Setup

- $|\mathcal{S}_{test}^{\theta}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their *triggering probability* and Trojans which are below some specific *triggering threshold* (P_{tr}) were selected. This constitutes our “hard-to-trigger” Trojan set.

Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{test}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their *triggering probability* and Trojans which are below some specific *triggering threshold* (P_{tr}) were selected. This constitutes our "hard-to-trigger" Trojan set.
- We set P_{tr} to be 10^{-6}.

Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{test}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their *triggering probability* and Trojans which are below some specific *triggering threshold* (P_{tr}) were selected. This constitutes our "hard-to-trigger" Trojan set.
- We set P_{tr} to be 10^{-6}.
- The whole scheme was implemented in C++ .

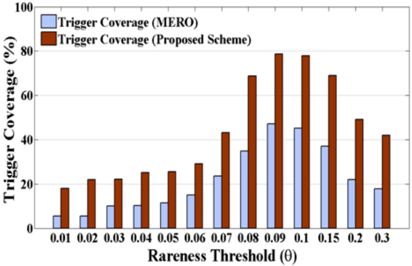
Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{test}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their *triggering probability* and Trojans which are below some specific *triggering threshold* (P_{tr}) were selected. This constitutes our "hard-to-trigger" Trojan set.
- We set P_{tr} to be 10^{-6}.
- The whole scheme was implemented in C++ .
- *Zchaff* [7] SAT solver was used.

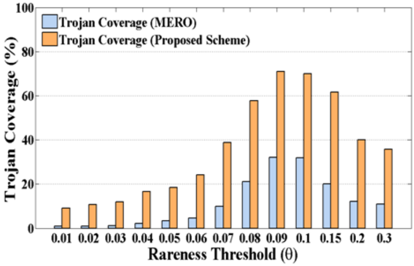
Experimental Results: Setup

- $|\mathcal{S}^{\theta}_{test}| = |\mathcal{S}| = 100000$ for each θ.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their *triggering probability* and Trojans which are below some specific *triggering threshold* (P_{tr}) were selected. This constitutes our "hard-to-trigger" Trojan set.
- We set P_{tr} to be 10^{-6}.
- The whole scheme was implemented in C++ .
- *Zchaff* [7] SAT solver was used.
- Sequential fault simulator *HOPE* [8] was used for fault simulation.

Experimental Results: circuit c7552

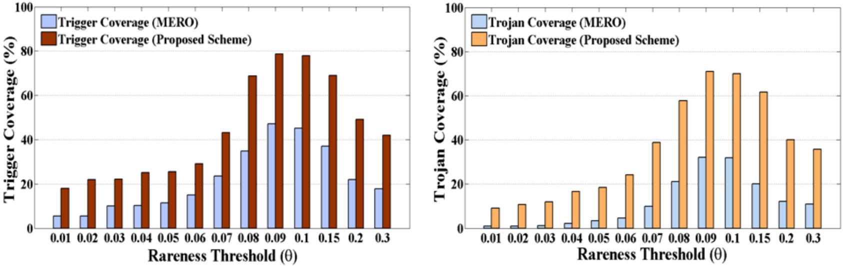


(a) Trigger Coverage



(b) Trojan Coverage

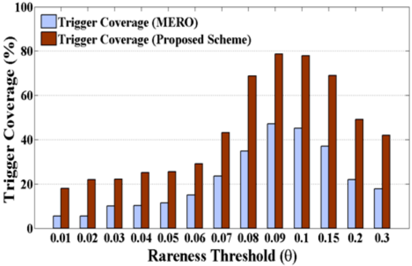
Experimental Results: circuit c7552



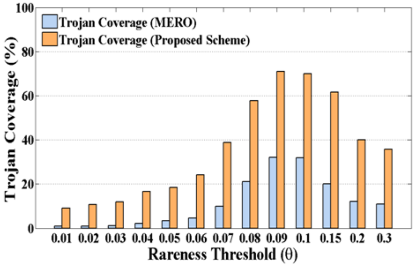
(a) Trigger Coverage

(b) Trojan Coverage

Experimental Results: circuit c7552



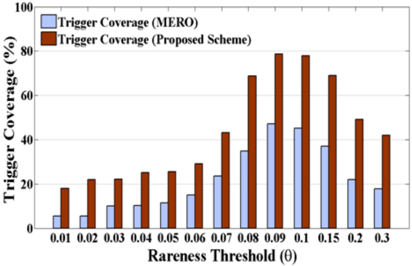
(a) Trigger Coverage



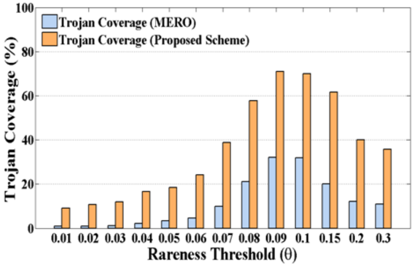
(b) Trojan Coverage

- Proposed scheme outperforms *MERO* to a significant extent.

Experimental Results: circuit c7552



(a) Trigger Coverage



(b) Trojan Coverage

- Proposed scheme outperforms *MERO* to a significant extent.
- The coverage trend is similar to *MERO* and the best operating point is 0.1.

Experimental Results on ISCAS Benchmarks

Table: Comparison of the proposed scheme with *MERO* with respect to testset length.

Ckt.	Gates	Testset (before Algo.-3)	Testset (after Algo.-3)	Testset (*MERO*)	Runtime (sec.)
c880	451	6674	5340	6284	9798.84
c2670	776	10,420	8895	9340	11299.74
c3540	1134	17,284	16,278	15,900	15720.19
c5315	1743	17,022	14,536	15,850	15877.53
c7552	2126	17,400	15,989	16,358	16203.02
s15850	9772	37,384	37,052	36,992	17822.67
s35932	16065	7849	7078	7343	14273.09
s38417	22179	53,700	50,235	52,735	19635.22

Experimental Results on ISCAS Benchmarks

Table: Comparison of the proposed scheme with *MERO* with respect to testset length.

Ckt.	Gates	Testset (before Algo.-3)	Testset (after Algo.-3)	Testset (*MERO*)	Runtime (sec.)
c880	451	6674	5340	6284	9798.84
c2670	776	10,420	8895	9340	11299.74
c3540	1134	17,284	16,278	15,900	15720.19
c5315	1743	17,022	14,536	15,850	15877.53
c7552	2126	17,400	15,989	16,358	16203.02
s15850	9772	37,384	37,052	36,992	17822.67
s35932	16065	7849	7078	7343	14273.09
s38417	22179	53,700	50,235	52,735	19635.22

Experimental Results on ISCAS Benchmarks

Table: Comparison of the proposed scheme with *MERO* with respect to testset length.

Ckt.	Gates	Testset (before Algo.-3)	Testset (after Algo.-3)	Testset (*MERO*)	Runtime (sec.)
c880	451	6674	5340	6284	9798.84
c2670	776	10,420	8895	9340	11299.74
c3540	1134	17,284	16,278	15,900	15720.19
c5315	1743	17,022	14,536	15,850	15877.53
c7552	2126	17,400	15,989	16,358	16203.02
s15850	9772	37,384	37,052	36,992	17822.67
s35932	16065	7849	7078	7343	14273.09
s38417	22179	53,700	50,235	52,735	19635.22

- Terminating condition of GA was set by the number of test vectors which *MERO* generates in is standard setup ($N = 1000$).

Experimental Results on ISCAS Benchmarks

Table: Comparison of the proposed scheme with *MERO* with respect to testset length.

Ckt.	Gates	Testset (before Algo.-3)	Testset (after Algo.-3)	Testset (*MERO*)	Runtime (sec.)
c880	451	6674	5340	6284	9798.84
c2670	776	10,420	8895	9340	11299.74
c3540	1134	17,284	16,278	15,900	15720.19
c5315	1743	17,022	14,536	15,850	15877.53
c7552	2126	17,400	15,989	16,358	16203.02
s15850	9772	37,384	37,052	36,992	17822.67
s35932	16065	7849	7078	7343	14273.09
s38417	22179	53,700	50,235	52,735	19635.22

- Terminating condition of GA was set by the number of test vectors which *MERO* generates in is standard setup ($N = 1000$).
- Sequential circuits were considered in full-scan mode.

Experimental Results on ISCAS Benchmarks

Table: Comparison of trigger and Trojan Coverage among *MERO* patterns and patterns generated with the proposed scheme with $\theta = 0.1$; $N = 1000$ (for *MERO*) and for trigger combinations containing up to four rare nodes.

Ckt.	*MERO*		Proposed Scheme	
	Trigger Coverage	Trojan Coverage	Trigger Coverage	Trojan Coverage
c880	75.92	69.96	96.19	85.70
c2670	62.66	49.51	87.15	75.82
c3540	55.02	23.95	81.55	60.00
c5315	43.50	39.01	85.91	71.13
c7552	45.07	31.90	77.94	69.88
s15850	36.00	18.91	68.18	57.30
s35932	62.49	34.65	81.79	73.52
s38417	21.07	14.41	56.95	38.10

Experimental Results on ISCAS Benchmarks

Table: Coverage comparison between *MERO* and the proposed Scheme for sequential Trojans.

Ckt.	Trig. Cov. for Proposed Scheme		Trig. Cov. for *MERO*	
	Trojan State Count		Trojan State Count	
	2	4	2	4
s15850	64.91	45.55	31.70	26.00
s35932	78.97	70.38	58.84	49.59
s38417	48.00	42.17	16.11	8.01

Ckt.	Troj. Cov. for Proposed Scheme		Troj. Cov. for *MERO*	
	Trojan State Count		Trojan State Count	
	2	4	2	4
s15850	46.01	32.59	13.59	8.95
s35932	65.22	59.29	25.07	15.11
s38417	30.52	19.92	9.06	2.58

Conclusion

- ATPG for Hardware Trojan detection is an important and less explored direction of research.

Conclusion

- ATPG for Hardware Trojan detection is an important and less explored direction of research.
- State-of-the-art techniques were not good enough.

Conclusion

- ATPG for Hardware Trojan detection is an important and less explored direction of research.
- State-of-the-art techniques were not good enough.
- Proposed scheme significantly improves the performance of the ATPG mechanism.

Conclusion

- ATPG for Hardware Trojan detection is an important and less explored direction of research.
- State-of-the-art techniques were not good enough.
- Proposed scheme significantly improves the performance of the ATPG mechanism.
- The generated Trojan database can be further used for Trojan diagnosis.

Conclusion

- ATPG for Hardware Trojan detection is an important and less explored direction of research.
- State-of-the-art techniques were not good enough.
- Proposed scheme significantly improves the performance of the ATPG mechanism.
- The generated Trojan database can be further used for Trojan diagnosis.
- Test vectors generated by the proposed scheme may also be utilized to improve the efficiency of side channel analysis based Trojan detection schemes.

References I

- DARPA, *TRUST in Integrated Circuits (TIC)*.
  [Online]. Available:
  http://www.darpa.mil/MTO/solicitations/baa07-24., 2007.

- M. Tehranipoor and F. Koushanfar, “A Survey of Hardware Trojan Taxonomy and Detection,” *Proc. of IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

- J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, “Security analysis of logic obfuscation,” in *Proceedings of the 49th Annual Design Automation Conference*, pp. 83–89, ACM, 2012.

- Y. Jin and Y. Makris, “Is single-scheme Trojan prevention sufficient?,” in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 305–308, IEEE, 2011.

References II

R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for hardware Trojan detection," in *Cryptographic Hardware and Embedded Systems-CHES 2009*, pp. 396–410, Springer, 2009.

H. Salmani, M. Tehranipoor, and J. Plusquellic, "New design strategy for improving hardware Trojan detection and reducing Trojan activation time," in *Proc. of Int. symposium on HOST*, pp. 66–73, 2009.

Z. Fu, Y. Marhajan, and S. Malik, "Zchaff sat solver," 2004.

References III

H. K. Lee and D. S. Ha, “HOPE: An efficient parallel fault simulator for synchronous sequential circuits,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 15, no. 9, pp. 1048–1058, 1996.

Questions?

Thank You...

Backup Slides

Experimental Results on ISCAS Benchmarks

Table: Trigger and Trojan coverage at various stages of the proposed scheme. at $\theta = 0.1$ for random sample of Trojans upto 4 rare node triggers (Sample size is $100,000$ for combinational circuits and $10,000$ for sequential circuits).

Ckt.	GA only		GA + SAT		GA + SAT + Algo. 3	
	Trig. Cov.	Troj. Cov.	Trig. Cov.	Troj. Cov.	Trig. Cov.	Troj. Cov.
c880	92.12	83.59	96.19	85.70	96.19	85.70
c2670	81.63	69.27	87.31	75.17	87.15	75.82
c3540	80.58	57.21	82.79	59.07	81.55	60.00
c5315	83.79	64.45	85.11	65.04	85.91	71.13
c7552	73.73	64.05	78.16	68.95	77.94	69.88
s15850	64.91	51.95	70.36	57.30	68.18	57.30
s35932	81.15	71.77	81.90	73.52	81.79	73.52
s38417	55.03	29.33	61.76	36.50	56.95	38.10

Probabilistic Analysis to find out Rare Nodes