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:

- 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
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.

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
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.
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:**

  - 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 have been done on Logic testing based Trojan detection.
Motivation

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

- 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.
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.
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.

Surprisingly, very few works has been done on Logic testing based Trojan detection.
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].

Surprisingly, very few works have been done on Logic testing based Trojan detection.
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].

Surprisingly, very few works has been done on Logic testing based Trojan detection.
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.
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.
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 their 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 their 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 their 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

- **Trigger inputs A and B**: internal rare nodes inside the circuit.
Logic Testing Based Trojan Detection: Trojan Models

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

Utilization:

- Rare nodes are selected based on a "rareness threshold" ($\theta$).
- **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.
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” $(\theta)$. 
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” \( \theta \).

**N-detect** ATPG scheme was proposed:
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” ($\theta$).

**$N$-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” ($\theta$).
- **$N$-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

- 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
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:
  - 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.
Genetic Algorithm and Boolean Satisfiability for ATPG

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

- 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.
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.

- 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.
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

Input: Circuit Netlist, Trojan sample size, rareness threshold ($\theta$), # GA parameters

Determine Rare Nodes by Probabilistic Analysis

Select Trojan Samples using Random Sampling ($S$)

Generate Test vectors with GA for the Trigger Patterns in ($S$) (Algorithm 1)

Generate Test vectors for the Patterns in $S$ unsolved by GA ($S^\prime$) using SAT (Algorithm 2)

Select Test Vectors which Propagates the Trojan Effect to the Output (Algorithm 3)

Compact set of Test Patterns ($T_{final}$)

Select Random Trojan Sample Sets with different $\theta$ values ($S^\theta_{test}$)

Generate Feasible Trojan Sample Sets ($S^{f}_{test}$)

Filter out Trojans below $P_n$ from each $S^{f}_{test}$ forming set $S^{\prime}_{test}$

Evaluate Effectiveness over $S^{\prime}_{test}$

END
**Proposed Scheme**

**Phase I**
- **Input:** Circuit Netlist, Trojan sample size, rareness threshold ($\theta$), # GA parameters
- **Determine Rare Nodes by Probabilistic Analysis**
- **Select Trojan Samples using Random Sampling ($S$)**
- **Generate Test vectors with GA for the Trigger Patterns in ($S$)** (Algorithm 1)

**Phase II**
- **Generate Test vectors for the Patterns in $S$ unsolved by GA ($S'$)** using SAT (Algorithm 2)

**Phase III**
- **Select Test Vectors which Propagates the Trojan Effect to the Output** (Algorithm 3)

**Validation**
- **Select Random Trojan Sample Sets with different $\theta$ values** ($S_{test}^\theta$)
- **Generate Feasible Trojan Sample Sets** ($S_{test}^{\ell}$)
- **Filter out Trojans below $P_\theta$ from each $S_{test}^{\ell}$ forming set $S_{test}^{\ell}$**
- **Evaluate Effectiveness over $S_{test}^{\ell}$**
- **END**

**Compact set of Test Patterns ($T_{final}$)**
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 number of test vectors has been generated.
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 number of test vectors has been generated.
Phase I: Genetic Algorithm

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

(a) rare 1
   rare 0
   rare 1

(b) rare 1
   rare 0
   rare 1

Satisfy Logic-1 here
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

Goal 1

- An effort to generate test vectors that would activate the most number of sampled trigger combinations.
## Phase I: Genetic Algorithm

<table>
<thead>
<tr>
<th>Goal 1</th>
<th>An effort to generate test vectors that would activate the most number of sampled trigger combinations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal 2</td>
<td>An effort to generate test vectors for hard-to-trigger combinations.</td>
</tr>
</tbody>
</table>
Phase I: Genetic Algorithm

Fitness Function

\[ f(t) = R \text{count}(t) + w \cdot I(t) \]  

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

Fitness Function

\[ f(t) = R_{count}(t) + w \times l(t) \]  

- \( f(t) \): 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 (\( > 1 \)).
- \( l(t) \): relative improvement of the database \( \mathcal{D} \) due to the test vector \( t \).
Phase I: Genetic Algorithm

Relative Improvement

\[ I(t) = n_2(s) - n_1(s) \]

\[ n_2(s) \]: number of test patterns in bins before update

\[ n_1(s) \]: number of test patterns in bins after update.
Phase I: Genetic Algorithm

Relative Improvement

\[ l(t) = \frac{n_2(s) - n_1(s)}{n_2(s)} \]  

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

\[ (D) \text{ with tuples } \{s, \{ti\} \}, \text{ with } s \in S \]

\[ \text{SAT}(s)? \]

Yes

\[ \{s, \{ti\}, \text{ where } s \in S_{\text{sat}} \} \]

No

\[ s \in S_{\text{unsat}} \]

\[ \text{Reject} \]

End

\[ |S'| = 0 \]
Phase II: Solving “Hard-to-Trigger” Patterns using SAT

\[ \{s, \varnothing\}, where \ s \in S' \]

\[ SAT(s) \]

\[ SAT Engine \]

\[ s \in S_{sat} \]

\[ s \in S_{unsat} \]

End

\[ \text{Is} \ |S'| = 0 \]

\[ \text{Trojan Database (D) with tuples } \{s, \{ti\}\}, \text{ with } s \in S \]

\[ \{s, \{ti\}\}, where \ s \in S_{sat} \]

\[ \text{Reject} \]
Phase II: Solving “Hard-to-Trigger” Patterns using SAT

- $S' \subseteq S$ denotes the set of trigger combinations unresolved by GA.
Phase II: Solving “Hard-to-Trigger” Patterns using SAT

- $S' \subseteq S$ denotes the set of trigger combinations unresolved by GA.
- $S_{sat} \subseteq S'$ is the set solved by SAT.
Phase II: Solving “Hard-to-Trigger” Patterns using SAT

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

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

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

An Example 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

An Example

(a) 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

An Example

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

An Example

Trojan is triggered by an input vector 1111.
Phase III: Payload Aware Test Vector Selection

An Example

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

- 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

\[ \{s, \{t_i\}\}, \text{where } s \in S_{\text{unsat}} \]

(1) \( |S_{\text{unsat}}| = 0 \)

(2) No

(3) Compute Pseudo Test Vector (PTV)

3-Value Logic Simulator

(4) Gen Fault List

(5) \( |\{t^s_i\}| > 5 \) ?

(6) Yes

HOPE

(6) \( \{F^s_i\} \)

(6) \( \{F^s_{\text{detected}}\} \subseteq \{F^s\} \)

(7) \( T^s_s \subseteq \{t^s_i\} \cup \{t^s_{ext}\} \)
Phase III: Pseudo Test Vector

For each set of test vectors \( \{t_s\} \) corresponding to a triggering combination \( s \), we find out the primary input positions which remain 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: Pseudo Test Vector

For each set of test vectors \( \{s_i\} \) corresponding to a triggering combination \( s \), we find out the primary input positions which remain 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: Pseudo Test Vector

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

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

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

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

1. Trojan Database ($D$) with tuples $\{s, \{t\}\}$, where $s \in S_{unsat}$

2. Is $|S_{unsat}| = 0$
   - Yes: End
   - No: $\{t_s^i\}$

3. Compute Pseudo Test Vector (PTV)

4. 3-Value Logic Simulator

5. $|\{t_s^i\}| > 5$
   - Yes: HOPE
   - No: Add Extra Vectors generated from PTV $\{t_s^i \cup t_{ext}^s\}$

6. Gen Fault List

7. $T_s \subseteq \{t_s^i\} \cup \{t_{ext}^s\}$
   - $F_s^{detected} \subseteq \{F_s\}$
Experimental Results: Setup

Input: Circuit Netlist, Trojan sample size, rareness threshold ($\theta$), # GA parameters

**Phase I**
- Determine Rare Nodes by Probabilistic Analysis
- Select Trojan Samples using Random Sampling ($S$)
- Generate Test vectors with GA for the Trigger Patterns in ($S'$) using SAT (Algorithm 1)

**Phase II**
- Generate Test vectors for the Patterns in $S$ unsolved by GA ($S'$) using SAT (Algorithm 2)

**Phase III**
- Select Test Vectors which Propagates the Trojan Effect to the Output (Algorithm 3)

**Validation**
- Select Random Trojan Sample Sets with different $\theta$ values ($S^{\theta}_{test}$)
- Generate Feasible Trojan Sample Sets ($S^{f}_{test}$)
- Filter out Trojans below $P_o$ from each $S^{f}_{test}$ forming set $S^{fr}_{test}$
- Evaluate Effectiveness over $S^{fr}_{test}$

Compact set of Test Patterns ($T_{final}$)

END
Experimental Results: Setup

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: Setup

- $|S_{test}^\theta| = |S| = 100000$ for each $\theta$. 

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: Setup

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

Zchaff [7] SAT solver was used. Sequential fault simulator HOPE [8] was used for fault simulation.
Experimental Results: Setup

- \(|S^\theta_{test}| = |S| = 100000\) for each \(\theta\).
- 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.

Zchaff \([7]\) SAT solver was used. Sequential fault simulator HOPE \([8]\) was used for fault simulation.
Experimental Results: Setup

- $|S^\theta_{\text{test}}| = |S| = 100000$ for each $\theta$.
- Feasible Trojans were selected from candidate Trojan set by extensive SAT solving and circuit simulation.
- Trojans were ranked according to their \textit{triggering probability} and Trojans which are below some specific \textit{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

- $|S^\theta_{test}| = |S| = 100000$ for each $\theta$.
- 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

- $|S^\theta_{\text{test}}| = |S| = 100000$ for each $\theta$.
- 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

- $|S^\theta_{test}| = |S| = 100000$ for each $\theta$.
- 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

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

(a) Trigger Coverage

(b) Trojan Coverage
Experimental Results: circuit c7552

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: circuit c7552

- Proposed scheme outperforms MERO to a significant extent.
Experimental Results: circuit c7552

- Proposed scheme outperforms MERO to a significant extent.
- The coverage trend is similar to MERO and the best operating point is 0.1.
**Table:** Comparison of the proposed scheme with *MERO* with respect to testset length.

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

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

<table>
<thead>
<tr>
<th>Ckt.</th>
<th>Gates</th>
<th>Testset (before Algo.-3)</th>
<th>Testset (after Algo.-3)</th>
<th>Testset (MERO)</th>
<th>Runtime (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c880</td>
<td>451</td>
<td>6674</td>
<td>5340</td>
<td>6284</td>
<td>9798.84</td>
</tr>
<tr>
<td>c2670</td>
<td>776</td>
<td>10,420</td>
<td>8895</td>
<td>9340</td>
<td>11299.74</td>
</tr>
<tr>
<td>c3540</td>
<td>1134</td>
<td>17,284</td>
<td>16,278</td>
<td>15,900</td>
<td>15720.19</td>
</tr>
<tr>
<td>c5315</td>
<td>1743</td>
<td>17,022</td>
<td>14,536</td>
<td>15,850</td>
<td>15877.53</td>
</tr>
<tr>
<td>c7552</td>
<td>2126</td>
<td>17,400</td>
<td>15,989</td>
<td>16,358</td>
<td>16203.02</td>
</tr>
<tr>
<td>s15850</td>
<td>9772</td>
<td>37,384</td>
<td>37,052</td>
<td>36,992</td>
<td>17822.67</td>
</tr>
<tr>
<td>s35932</td>
<td>16065</td>
<td>7849</td>
<td>7078</td>
<td>7343</td>
<td>14273.09</td>
</tr>
<tr>
<td>s38417</td>
<td>22179</td>
<td>53,700</td>
<td>50,235</td>
<td>52,735</td>
<td>19635.22</td>
</tr>
</tbody>
</table>
Table: Comparison of the proposed scheme with \textit{MERO} with respect to testset length.

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

Terminating condition of GA was set by the number of test vectors which \textit{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.

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

- 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.
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.

<table>
<thead>
<tr>
<th>Ckt.</th>
<th>MERO</th>
<th>Proposed Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trigger Coverage</td>
<td>Trojan Coverage</td>
</tr>
<tr>
<td>c880</td>
<td>75.92</td>
<td>69.96</td>
</tr>
<tr>
<td>c2670</td>
<td>62.66</td>
<td>49.51</td>
</tr>
<tr>
<td>c3540</td>
<td>55.02</td>
<td>23.95</td>
</tr>
<tr>
<td>c5315</td>
<td>43.50</td>
<td>39.01</td>
</tr>
<tr>
<td>c7552</td>
<td>45.07</td>
<td>31.90</td>
</tr>
<tr>
<td>s15850</td>
<td>36.00</td>
<td>18.91</td>
</tr>
<tr>
<td>s35932</td>
<td>62.49</td>
<td>34.65</td>
</tr>
<tr>
<td>s38417</td>
<td>21.07</td>
<td>14.41</td>
</tr>
</tbody>
</table>
### Experimental Results on ISCAS Benchmarks

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

<table>
<thead>
<tr>
<th>Ckt.</th>
<th>Trig. Cov. for Proposed Scheme</th>
<th>Trig. Cov. for <em>MERO</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trojan State Count</td>
<td>Trojan State Count</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>s15850</td>
<td>64.91</td>
<td>45.55</td>
</tr>
<tr>
<td>s35932</td>
<td>78.97</td>
<td>70.38</td>
</tr>
<tr>
<td>s38417</td>
<td>48.00</td>
<td>42.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ckt.</th>
<th>Trig. Cov. for Proposed Scheme</th>
<th>Trig. Cov. for <em>MERO</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trojan State Count</td>
<td>Trojan State Count</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>s15850</td>
<td>46.01</td>
<td>32.59</td>
</tr>
<tr>
<td>s35932</td>
<td>65.22</td>
<td>59.29</td>
</tr>
<tr>
<td>s38417</td>
<td>30.52</td>
<td>19.92</td>
</tr>
</tbody>
</table>
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.
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.
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.
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.


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).

<table>
<thead>
<tr>
<th>Ckt</th>
<th>GA only</th>
<th></th>
<th>GA + SAT</th>
<th></th>
<th>GA + SAT + Algo. 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c880</td>
<td>92.12</td>
<td>83.59</td>
<td>96.19</td>
<td>85.70</td>
<td>96.19</td>
<td>85.70</td>
</tr>
<tr>
<td>c2670</td>
<td>81.63</td>
<td>69.27</td>
<td>87.31</td>
<td>75.17</td>
<td>87.15</td>
<td>75.82</td>
</tr>
<tr>
<td>c3540</td>
<td>80.58</td>
<td>57.21</td>
<td>82.79</td>
<td>59.07</td>
<td>81.55</td>
<td>60.00</td>
</tr>
<tr>
<td>c5315</td>
<td>83.79</td>
<td>64.45</td>
<td>85.11</td>
<td>65.04</td>
<td>85.91</td>
<td>71.13</td>
</tr>
<tr>
<td>c7552</td>
<td>73.73</td>
<td>64.05</td>
<td>78.16</td>
<td>68.95</td>
<td>77.94</td>
<td>69.88</td>
</tr>
<tr>
<td>s15850</td>
<td>64.91</td>
<td>51.95</td>
<td>70.36</td>
<td>57.30</td>
<td>68.18</td>
<td>57.30</td>
</tr>
<tr>
<td>s35932</td>
<td>81.15</td>
<td>71.77</td>
<td>81.90</td>
<td>73.52</td>
<td>81.79</td>
<td>73.52</td>
</tr>
<tr>
<td>s38417</td>
<td>55.03</td>
<td>29.33</td>
<td>61.76</td>
<td>36.50</td>
<td>56.95</td>
<td>38.10</td>
</tr>
</tbody>
</table>
Probabilistic Analysis to find out Rare Nodes