GATE Laboratory GMU

SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks

Conference on Cryptographic Hardware and Embedded Systems 2019 (**CHES 2019**)

Kimia Zamiri Azar, Hadi Mardani Kamali,
Houman Homayoun, and Avesta Sasan

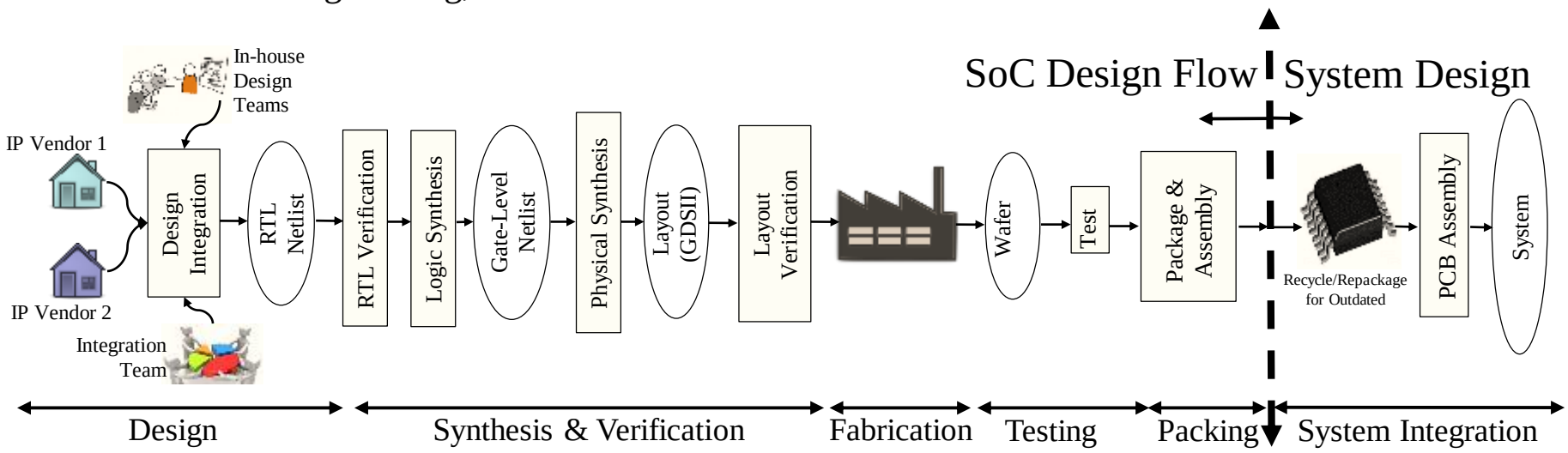
Department of Electrical and Computer Engineering
George Mason University, USA.

Outline

- Intro to Hardware Security
- Intro to Logic Locking
- SAT Attack and its Limitations
- SMT attack
  - SMT reduced to SAT Attack
  - Eager SMT Attack
  - Lazy SMT Attack
  - Accelerated Lazy SMT Attack
- Experimental Results
- Conclusion

Design Flow

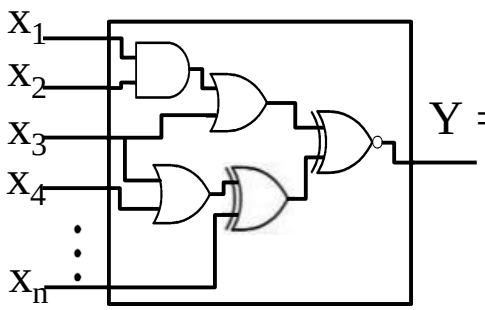
- High **Cost** of Manufacturing in ASIC Design has pushed most of needed fabrication offshore
  - Some Fabs are untrusted
- **Security threats** for untrusted supply chain
  - Trojan Insertion
  - Overproduction
  - Intellectual Property (IP) Theft
  - Counterfeiting
  - Reverse Engineering, etc.



Logic Locking

- **Logic Locking**: Adding Ambiguity to the Design
  - Inserting Key Programmable Gates (**KPGs**)
  - No Information on Key at Untrusted Entities

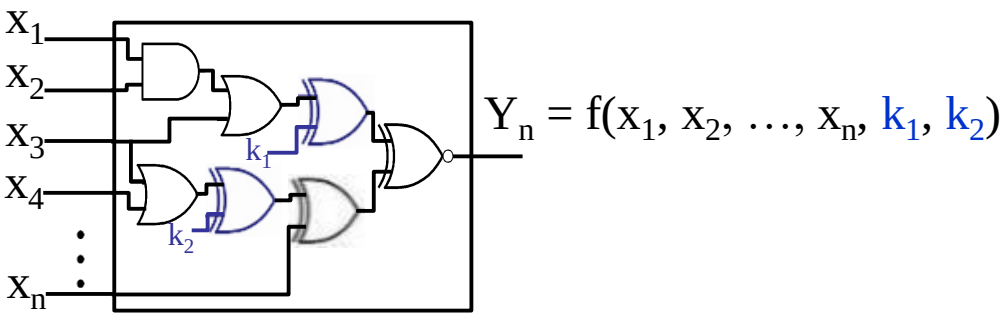
Original Netlist



Logic Locking

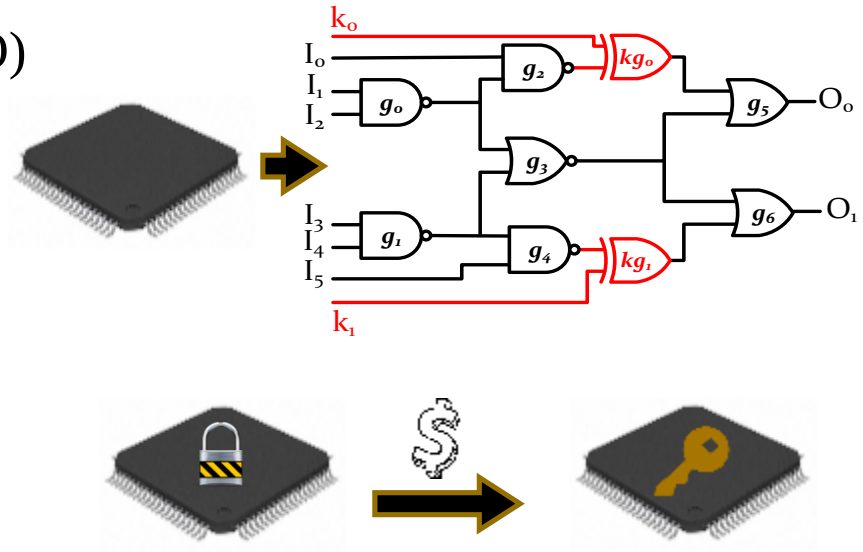
EPIC (2008)

Random Insertion Policy (RLL)



SAT Attack: a Turning Point in Logic Locking

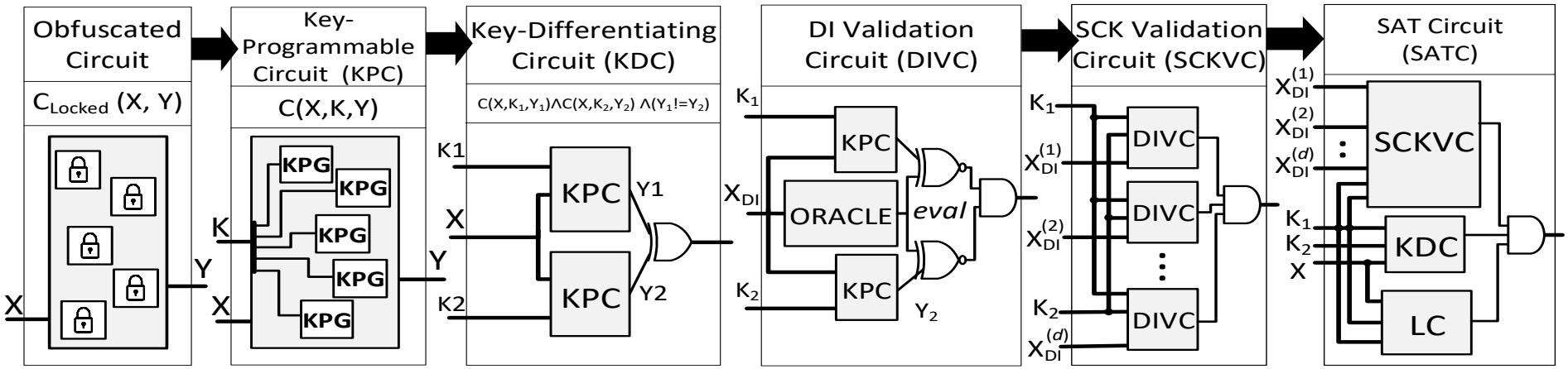
- SAT Attack Recipe:
  1. Reverse-engineered netlist (CL)
  2. A functionally activated chip (CO)

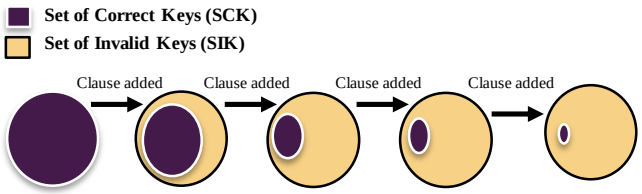


- SAT attack broke all logic obfuscation scheme prior to its debut!
  - Random insertion (RLL)
  - Fault-analysis (FLL)
  - Interference-based logic locking (SLL)

SAT Attack

- SAT Attack





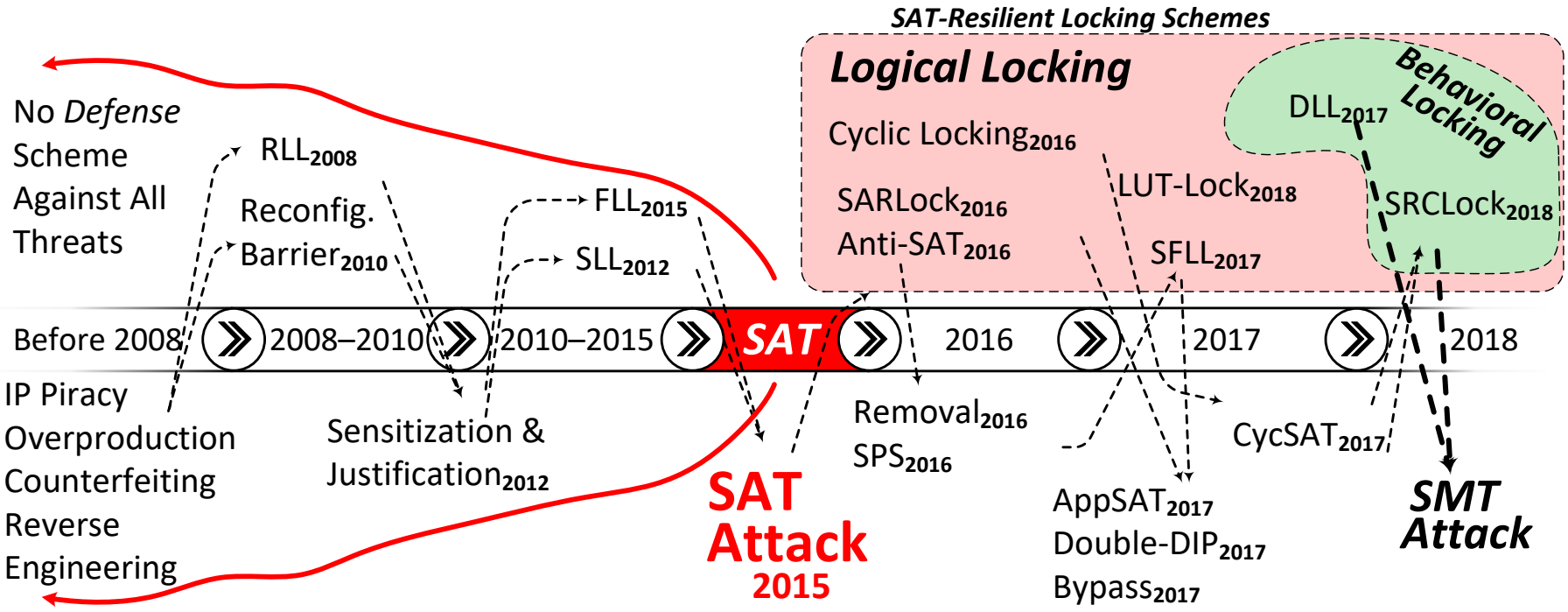
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

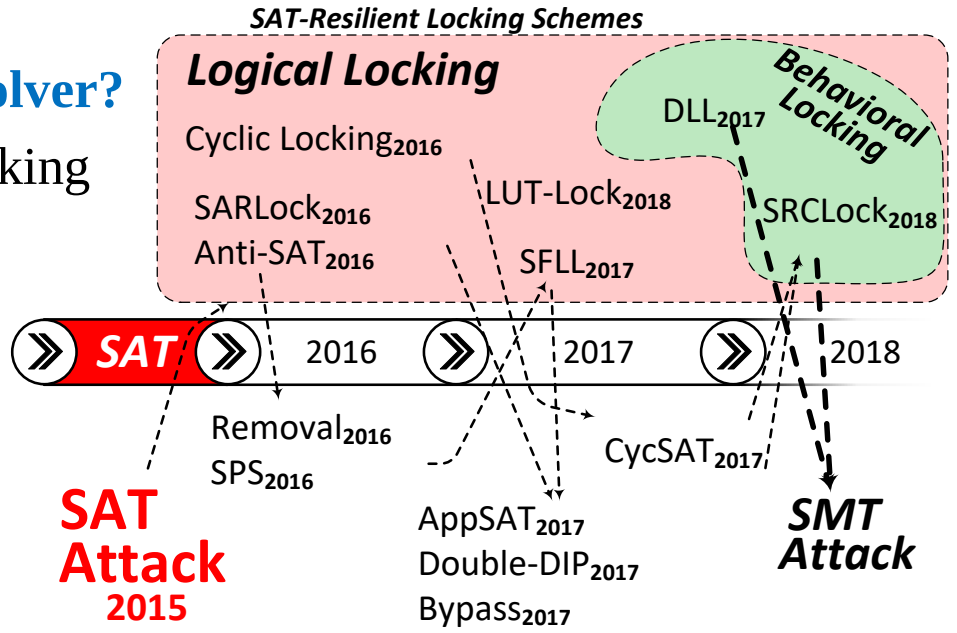
Limitation of SAT Attack

- SAT-Resilient Logic Obfuscation Solutions



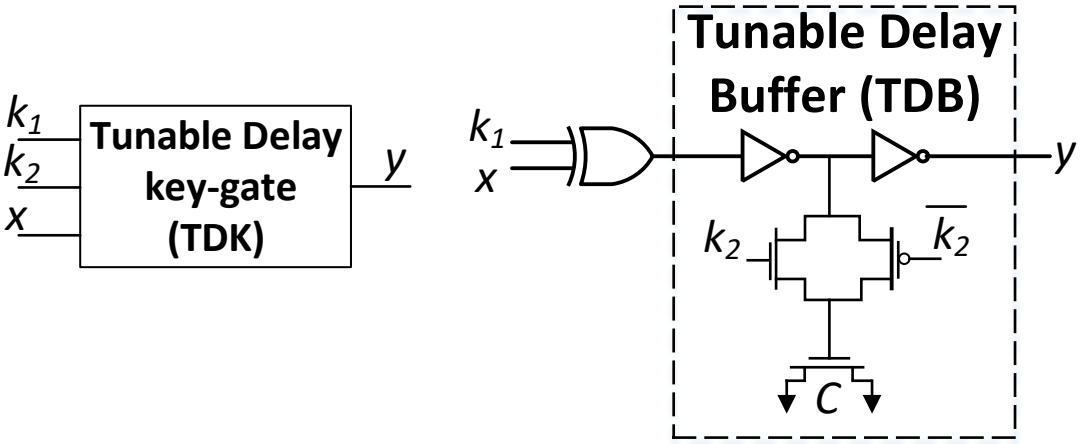
Limitation of SAT Attack

- A SAT Attack works if Logic obfuscation is of **Boolean nature**
- Model Translation Flow:
  - **Boolean** logic ➔ Conjunctive Normal Form (**CNF**)
  - **CNF** ➔ **Satisfiability** assignment problem
- Defense solutions to **trap the SAT solver?**
  - Use **non-logical** properties for locking
  - Can not be modeled if **could not be translated to CNF**



Behavioral logical obfuscation

- Delay and Logic Locking (DLL)
  - **Obfuscation** control the setup and hold
  - **Incorrect** key ➔ Setup and Hold time **violation**
  - Timing is **not translatable** to CNF
    - SAT solver remains **oblivious** to the keys used for timing obfuscation



$k_1 k_2$	f(x)	Delay
00	$y = x$	d_0
01	$y = x$	d_1
10	$y = \overline{x}$	d_0
11	$y = \overline{x}$	d_1

Solution

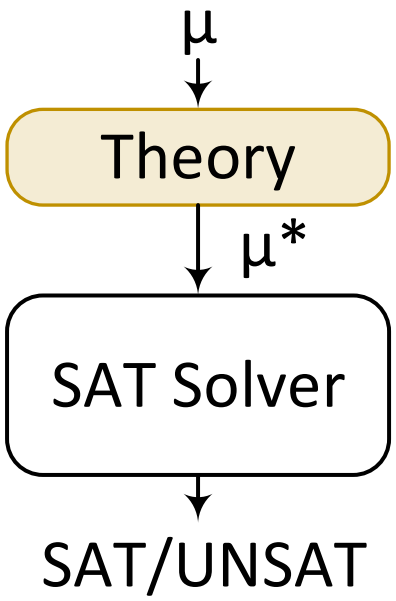
Satisfiability Modulo Theory (SMT) Attack

SMT Solver

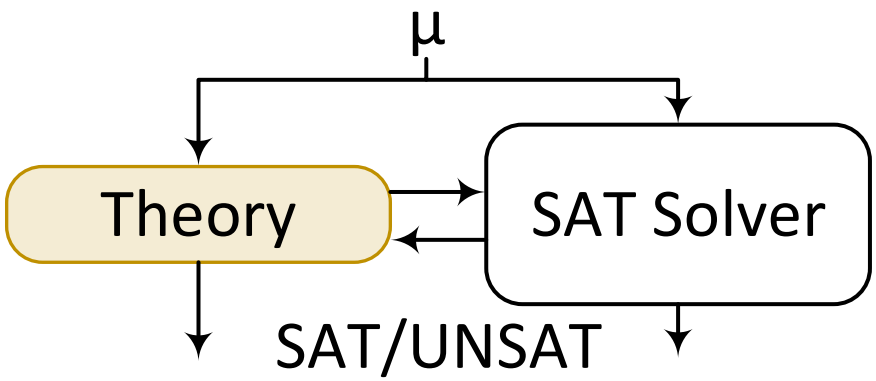
- A **SMT** is used to solve a **decision problem**
- Close **integration** of a **SAT** solver with **Theory** solver
- Uses **first-order theories**
  - Equality
  - Reasoning
  - Arithmetic
  - Graph-based deduction
- Modern SMT solvers provide the capability
  - **Combining** theory solvers
  - Can support **more powerful languages** as its input

Approaches to SMT Solver

- **Two approaches** for solving an SMT problem
  - **Eager** approach
  - **Lazy** approach



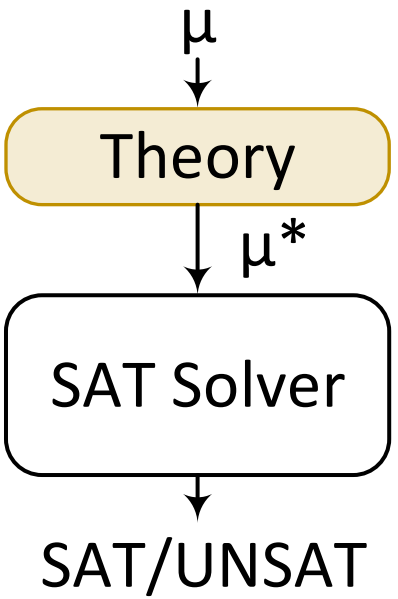
Eager approach



Lazy approach

SMT Eager Approach

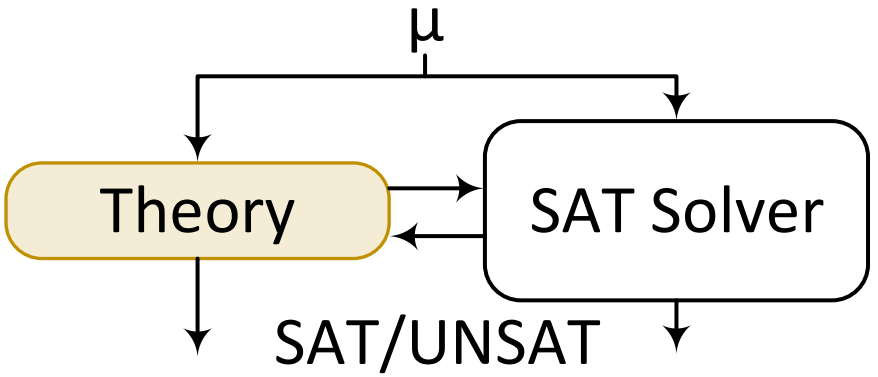
- **Eager** approach
  - **Translating** the problem into a Boolean SAT instances
  - The existing Boolean **SAT solvers** are **used as is**
  - The SMT **solver** has to **work** a lot **harder**
    - e.g. for checking the equivalence of two 32-bit values
  - By deploying a theory solver
    - this could be achieved in no time



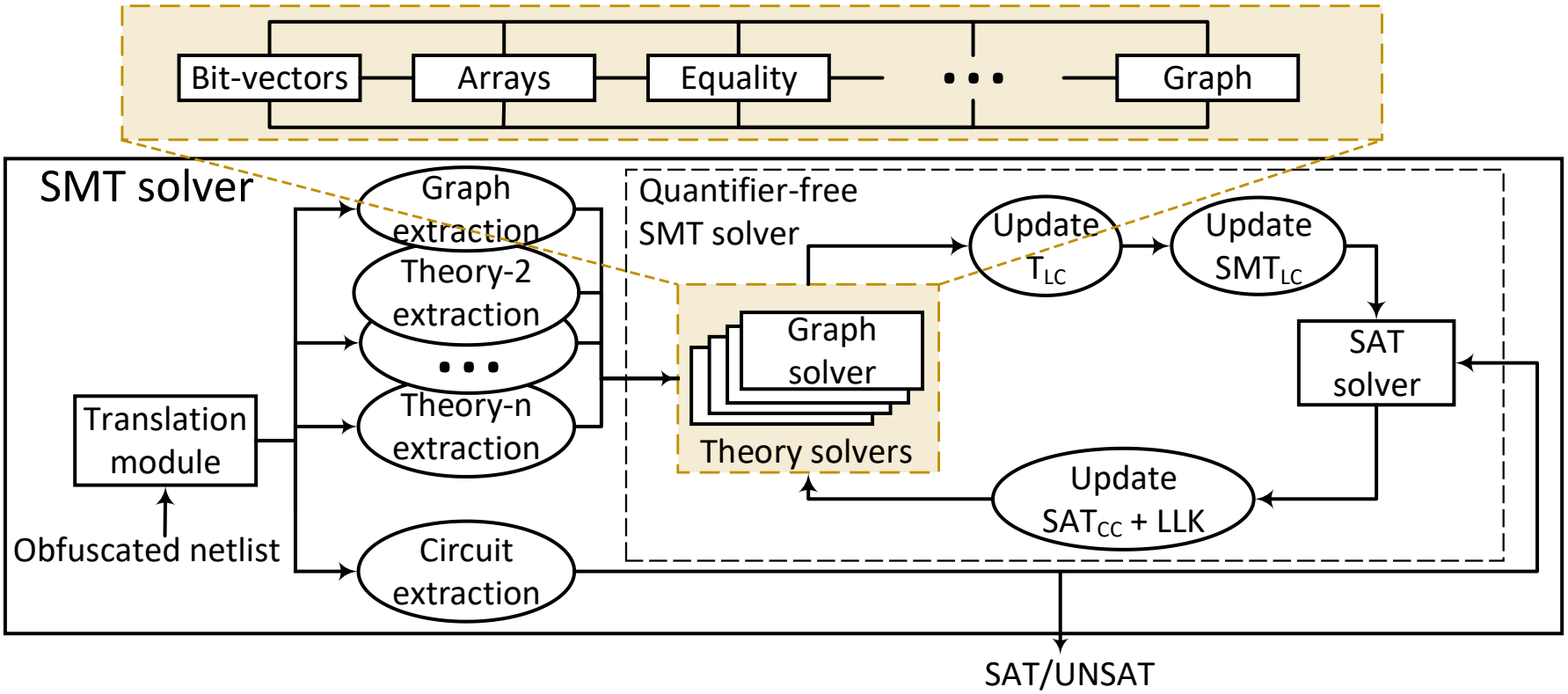
SMT Lazy Approach

- **Lazy** approach
  - **Integrates** the Boolean **satisfiability** solvers and **theory** solvers

- **Capabilities** of the Theory solvers:
  - **Theory propagation**
    - for checking possible conflicts on partial assignments
  - **Clause learning**
    - to speed-up pruning the decision tree.

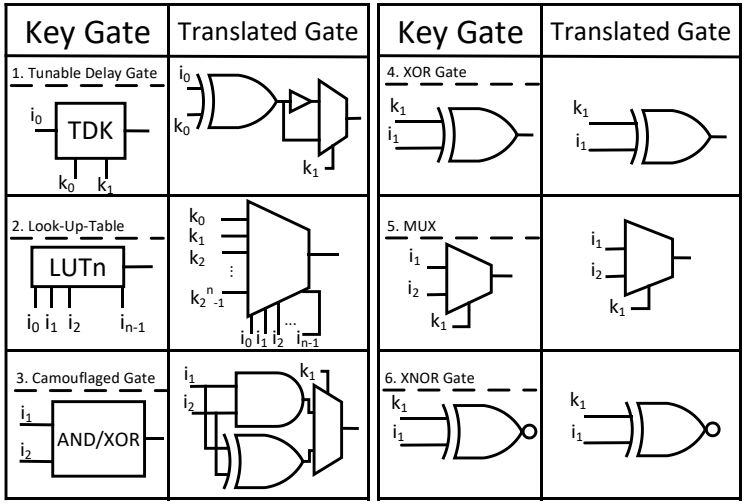


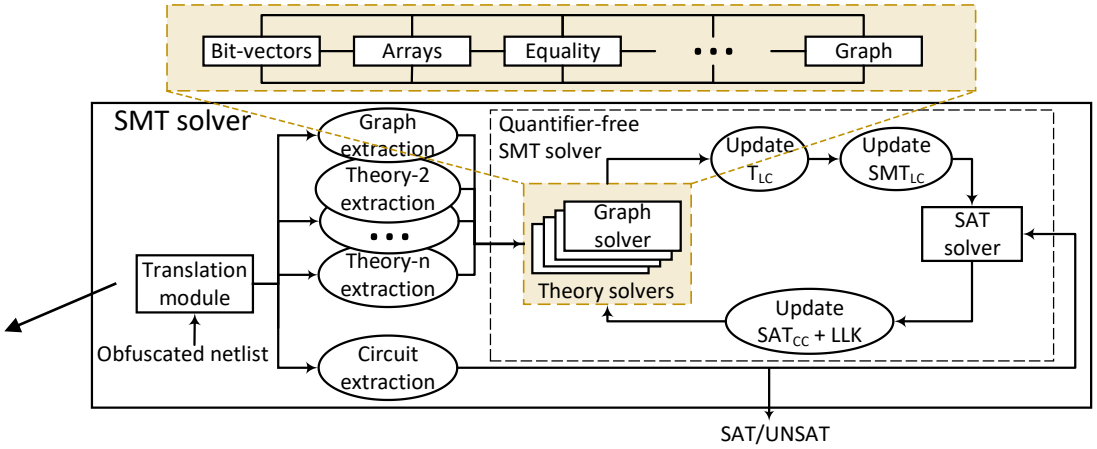
SMT Attack


Bit-vectors
Arrays
Equality
Graph
SMT solver
Graph extraction
Theory-2 extraction
Theory-n extraction
Translation module
Obfuscated netlist
Circuit extraction
Quantifier-free SMT solver
Graph solver
Theory solvers
Update T_{LC}
Update SMT_{LC}
SAT solver
Update SAT_{CC} + LLK
SAT/UNSAT

SMT Attack

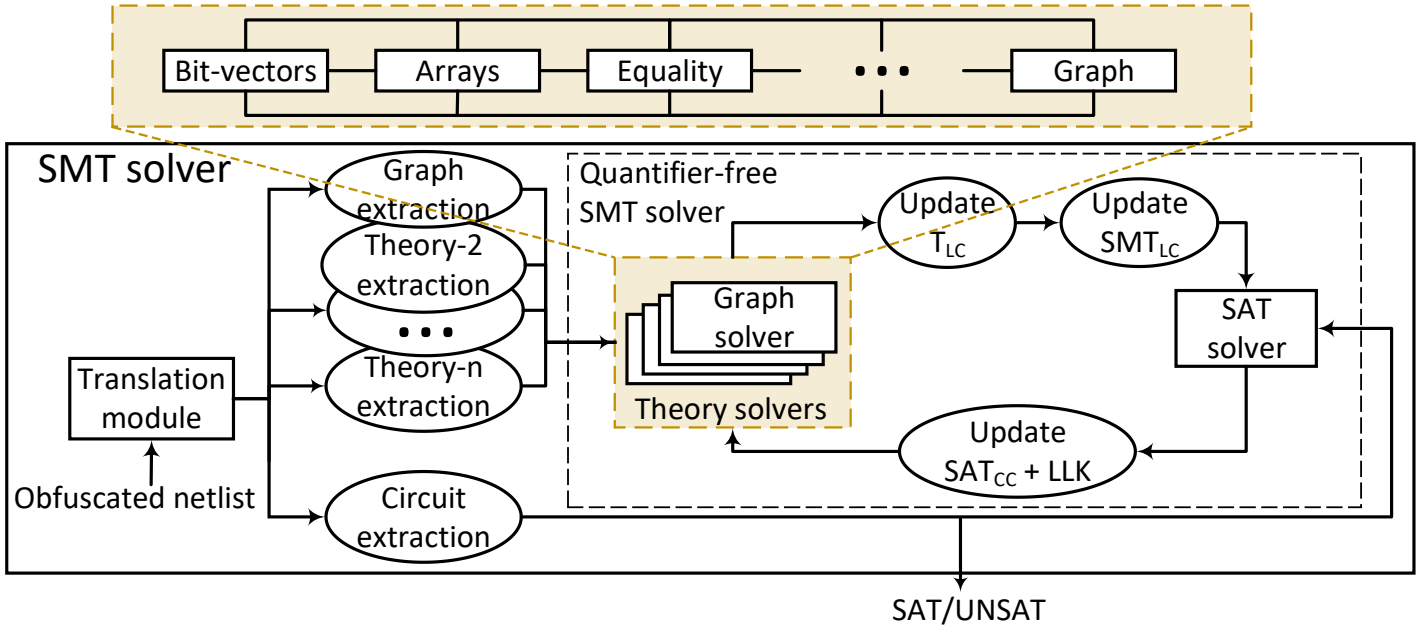
- **Step 1**
  - **Obfuscated cells** ➔ equivalent Key Programmable Gates (**KPG**)
  - A **KPG**
    - performs the same function as the obfuscated cell
    - allows building a key controlled representation





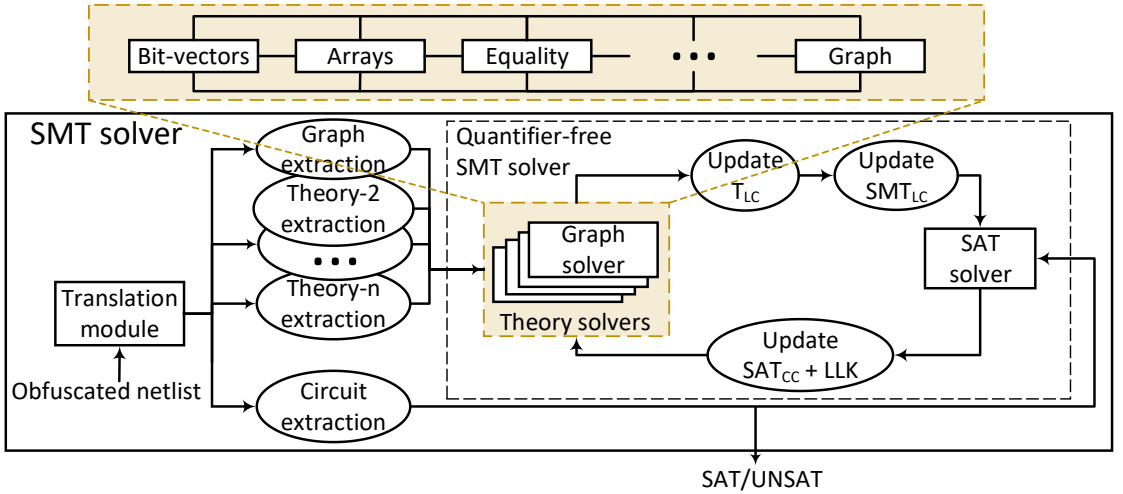
SMT Attack

- **Step 2**
  - Before invoking a theory solver
    - Input **model** ➔ model which is **understood by that theory solver**
    - **Different** translation step **for each theory** solver



SMT Attack

- Invoking the **SMT** solver **returns**
  - A **satisfiable assignment**
  - list of **learned theory**
  - **conflict clauses**



Attack Modes

- **Mode 1**: SMT **reduced** to SAT Attack
  - To show SMT is a superset of SAT
- **Mode 2**: **Eager** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Serialized!
- **Mode 3**: **Lazy** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Parallelized!
- **Mode 4**: **Accelerated** Lazy SMT Attack (AccSMT)
  - To show more efficiency
  - Uses BitVector Theory Solver

Attack Modes

- **Mode 1**: SMT **reduced** to SAT Attack
  - To show SMT is a superset of SAT
- **Mode 2**: **Eager** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Serialized!
- **Mode 3**: **Lazy** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Parallelized!
- **Mode 4**: **Accelerated** Lazy SMT Attack (AccSMT)
  - To show more efficiency
  - Uses BitVector Theory Solver

Mode 1: SMT reduced to SAT Attack

- SMT solver is a **superset** of SAT solver
  - Any attack formulated for SAT ➔ can be formulated using SMT

- **one-to-one translation** of the original SAT attack

Algorithm SMT Reduced to SAT Attack

function SAT_ATTACK(Obfuscated_Netlist N_{obf}, Functional_Circuit C_{org})
  $KPC \leftarrow$ ReplaceKPG(N_{obf});
  $C(X, K, Y) \leftarrow$ Circuit_Translation_to_CNF(KPC);
  $KDC = C(X, K_1, Y_1) \wedge C(X, K_2, Y_2) \wedge (Y_1 \neq Y_2)$;
  $SCKVC = TRUE$;
  $SATC = KDC \wedge SCKVC$;
  $LC = TRUE$; ▷ Learned Clauses
  $SMT_{LC} \leftarrow SATC$;
  while $(((X_{DI}, K_1, K_2, CC) \leftarrow SMT.Solve(SMT_{LC})) = TRUE)$ **do**
    $Y_f \leftarrow C_{org}(X_{DI})$;
    $DIVC = C(X_{DI}, K_1, Y_f) \wedge C(X_{DI}, K_2, Y_f)$;
    $SCKVC = SCKVC \wedge DIVC$;
    $LC = LC \wedge CC$
    $SMT_{LC} = KDC \wedge SCKVC \wedge LC$;
  $Key \leftarrow SMT.Solve(SMT_{LC})$;

Mode 1: SMT reduced to SAT Attack

- The recently found **Conflict Clauses (CC)** are **added** to the set of previously found **Learned Clauses (LC).**

- Note that this step is done **implicitly if** SMT is **stateful**.

Algorithm SMT Reduced to SAT Attack

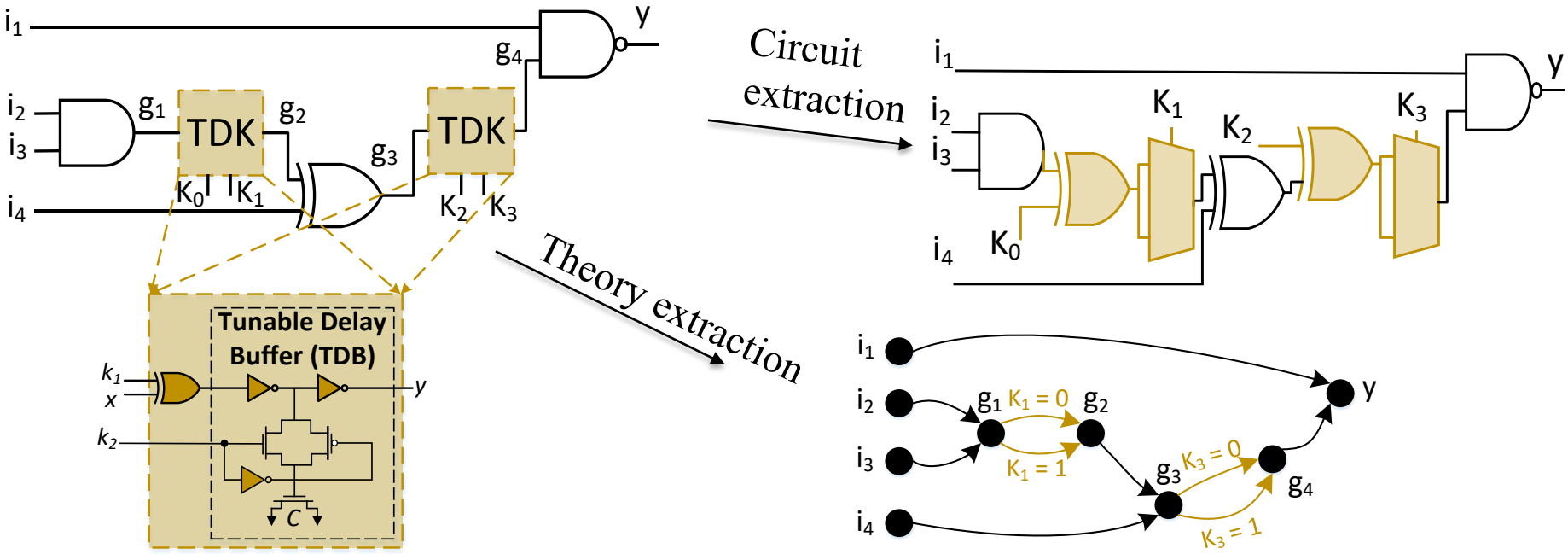
```
1:  function SAT_ATTACK(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X, K, Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                   ▷ Learned Clauses
8:      SMT_LC ← SATC;
9:      while (((X_DI, K1, K2, CC) ← SMT.Solve(SMT_LC))= TRUE) do
10:         Y_f ← C_org(X_DI);
11:         DIVC = C(X_DI, K1, Y_f) ∧ C(X_DI, K2, Y_f);
12:         SCKVC = SCKVC ∧ DIVC;
13:         LC = LC ∧ CC
14:         SMT_LC = KDC ∧ SCKVC ∧ LC;
15:     Key ← SMT.Solve(SMT_LC);
```

Attack Modes

- **Mode 1**: SMT **reduced** to SAT Attack
  - To show SMT is a superset of SAT
- **Mode 2**: **Eager** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Serialized!
- **Mode 3**: **Lazy** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Parallelized!
- **Mode 4**: **Accelerated** Lazy SMT Attack (AccSMT)
  - To show more efficiency
  - Uses BitVector Theory Solver

Case Study

- ***Case Study***: Delay and Logic Locking (DLL) *[1]

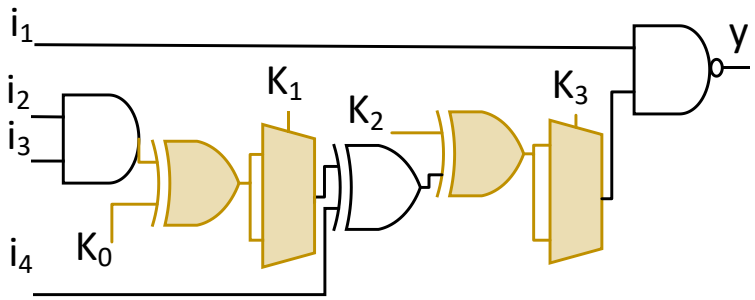


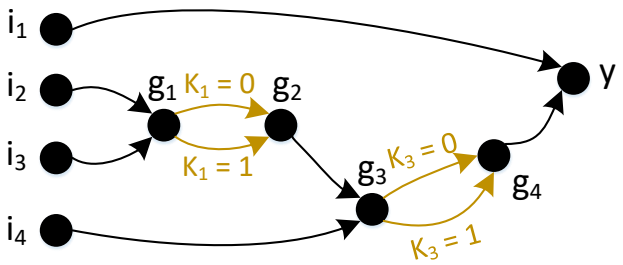
*[1] Y. Xie and A. Srivastava, "*Delay Locking: Security Enhancement of Logic Locking against IC Counterfeiting and Overproduction,*" In Proceedings of the 54th Annual Design Automation Conference (**DAC'17**), 2017.

Case Study

- ***Case Study***: Delay and Logic Locking (DLL) *[1]
- K_1 and K_3
  - **No impact** on the **logical behavior** of the circuit
  - Only **changes** its **delay**

- **SAT attack** results
  - **Random assignment** to K_1 and K_3





*[1] Y. Xie and A. Srivastava, "*Delay Locking: Security Enhancement of Logic Locking against IC Counterfeiting and Overproduction,*" In Proceedings of the 54th Annual Design Automation Conference (**DAC'17**), 2017.

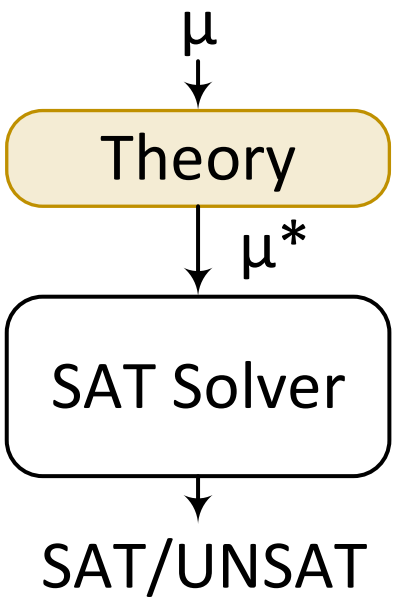
Mode 2: Eager SMT Attack

Algorithm Eager SMT Attack on DLL

```
1:  function SMT_Eager_Att(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                          ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_LC ← GenTLC(G(X,K));                              ▷ Theory Learned Clauses
10:     SMT_LC ← SATC ∧ T_LC;                               ▷ SMT Clauses
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC);
```

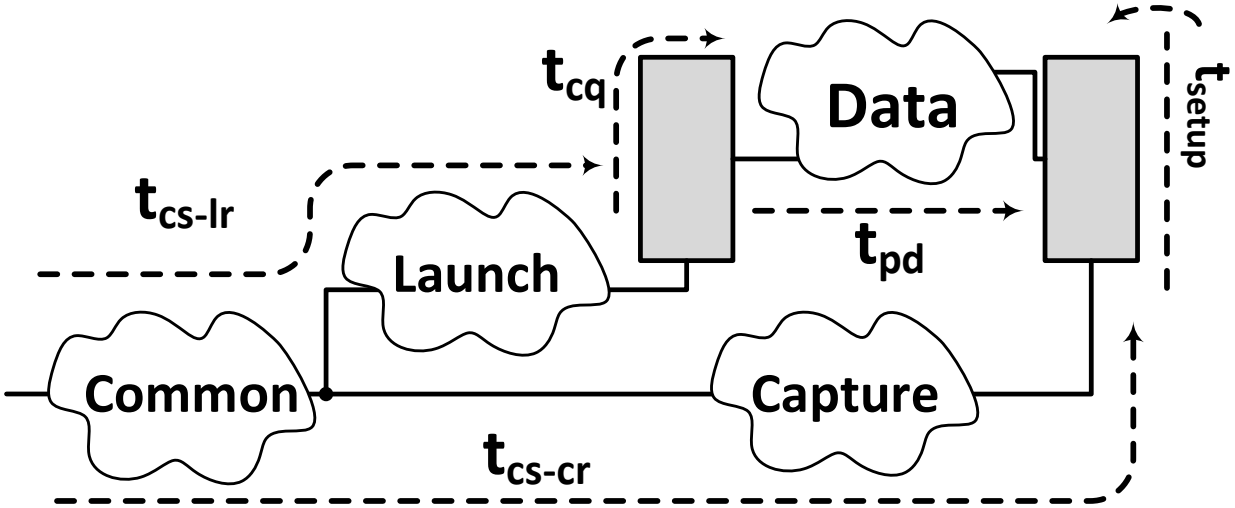
Pre-Processing step by using a graph theory solver for SMT attack (*Eager*)

```
1:  function GenTLC(Graph G)
2:      Inputs ← G.find_start_points();
3:      Outputs ← G.find_end_points();
4:      T_LC ← []
5:      for each (Sp in Inputs) do
6:          for each (Ep in Outputs) do
7:              Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:              Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:              T_LC ← SMT.solve(Upper(Sp,Ep)(K) ∧ Lower(Sp,Ep)(K) ∧ T_LC);
10:     return T_LC
```



Mode 2: Eager SMT Attack

- Calculating Hold Time and Setup Time



$$t_{cs-lr} + t_{clk-q} + t_p + t_{setup} + U \leq t_{cs-cr} + T_{clk}$$

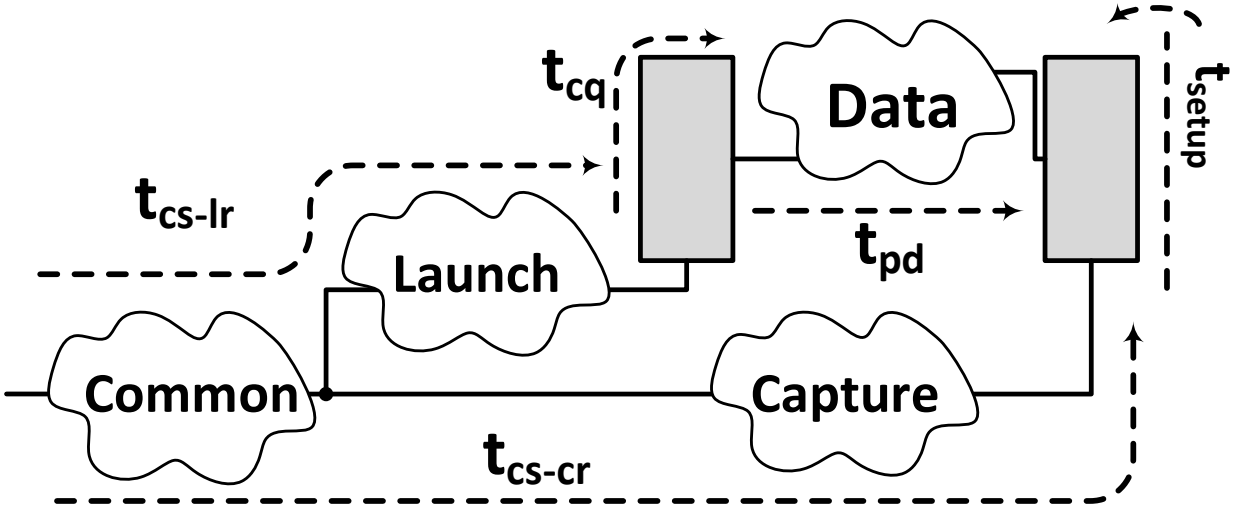
$$t_{cs-lr} + t_{clk-q} + t_{cd} \geq t_{hold} + t_{cs-cr} + U$$

$$\mathbf{t}_p \leq T_{clk} + (t_{cs-cr} - t_{cs-lr}) - t_{clk-q} - t_{setup} - U = Upper$$

$$\mathbf{t}_{cd} \geq t_{hold} + (t_{cs-cr} - t_{cs-lr}) - t_{clk-q} + U = Lower$$

Mode 2: Eager SMT Attack

- Calculating Hold Time and Setup Time



$$t_{cs-lr} + t_{clk-q} + t_p + t_{setup} + U \leq t_{cs-cr} + T_{clk}$$

$$t_{cs-lr} + t_{clk-q} + t_{cd} \geq t_{hold} + t_{cs-cr} + U$$

$$\mathbf{t}_p \leq T_{clk} + (t_{cs-cr} - t_{cs-lr}) - t_{clk-q} - t_{setup} - U = Upper$$

$$\mathbf{t}_{cd} \geq t_{hold} + (t_{cs-cr} - t_{cs-lr}) - t_{clk-q} + U = Lower$$

Mode 2: Eager SMT Attack

Algorithm Eager SMT Attack on DLL

```
1:  function SMT_Eager_Att(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                          ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_LC ← GenTLC(G(X,K));                              ▷ Theory Learned Clauses
10:     SMT_LC ← SATC ∧ T_LC;                               ▷ SMT Clauses
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC);
```

Pre-Processing step by using a graph theory solver for SMT attack (*Eager*)

```
1:  function GenTLC(Graph G)
2:      Inputs ← G.find_start_points();
3:      Outputs ← G.find_end_points();
4:      T_LC ← []
5:      for each (Sp in Inputs) do
6:          for each (Ep in Outputs) do
7:              Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:              Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:              T_LC ← SMT.solve(Upper(Sp,Ep)(K) ∧ Lower(Sp,Ep)(K) ∧ T_LC);
10:     return T_LC
```

Mode 2: Eager SMT Attack

Algorithm Eager SMT Attack on DLL

```
1:  function SMT_Eager_Att(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                          ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_LC ← GenTLC(G(X,K));                              ▷ Theory Learned Clauses
10:     SMT_LC ← SATC ∧ T_LC;                               ▷ SMT Clauses
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC);
```

Pre-Processing step by using a graph theory solver for SMT attack (*Eager*)

```
1:  function GenTLC(Graph G)
2:      Inputs ← G.find_start_points();
3:      Outputs ← G.find_end_points();
4:      T_LC ← []
5:      for each (Sp in Inputs) do
6:          for each (Ep in Outputs) do
7:              Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:              Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:              T_LC ← SMT.solve(Upper(Sp,Ep)(K) ∧ Lower(Sp,Ep)(K) ∧ T_LC);
10:     return T_LC
```

Limitation of Eager SMT Attack

- **For some problems the Eager approach does not work!**
  - **Why?** Eager relies on **reduction** of a problem to a SAT problem

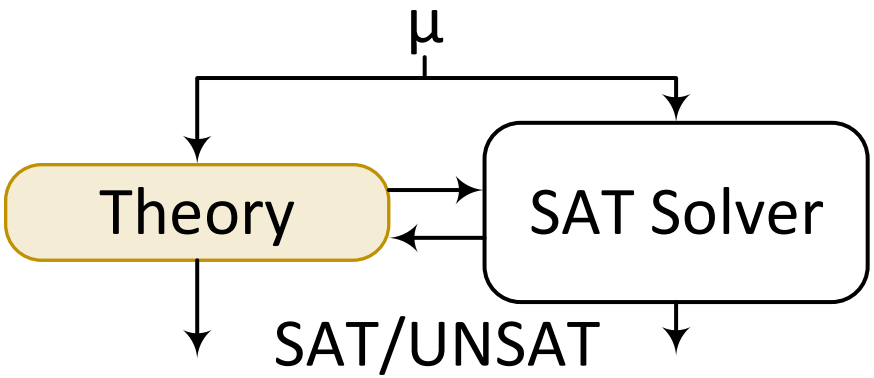
- **SRCLock**
  - # of cycles is exponential w.r.t. the # of inserted feedbacks
  - The run time of pre-processing is exponential
    - w.r.t. the # of inserted feedbacks
  - Preventing us to ever reach the SAT attack

Attack Modes

- **Mode 1**: SMT **reduced** to SAT Attack
  - To show SMT is a superset of SAT
- **Mode 2**: **Eager** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Serialized!
- **Mode 3**: **Lazy** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Parallelized!
- **Mode 4**: **Accelerated** Lazy SMT Attack (AccSMT)
  - To show more efficiency
  - Uses BitVector Theory Solver

Mode 3: Lazy SMT Attack

- **Lazy** approach of SMT attack
  - Moves from **pre-processing** to **co-processing**



Mode 3: Lazy SMT Attack

Algorithm Overall SMT Attack (*Lazy* Approach)

```
1:  function SMT_LAZY_ATT(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                   ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_CE(K) ← GenTCE(G(X, K));        ▷ Theory Constraint Expressions (Not Solved)
10:     T_CE(K1,K2) ← T_CE(K1) ∪ T_CE(K2) ;
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SATC, T_CE(K1,K2)))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC,T_CE(K));
```

Initialization of constraints for SMT attack (*Lazy* Approach)

```
1:      function GENTCE(Graph G)
2:          Inputs ← G.find_start_points();
3:          Outputs ← G.find_end_points();
4:          T_CE(K) ← []
5:          for each (Sp in Inputs) do
6:              for each (Ep in Outputs) do
7:                  Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:                  Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:                  Range(Sp,Ep)(K) ← Lower(Sp,Ep)(K) ∧ Upper(Sp,Ep)(K);
10:                 T_CE(K) ← T_CE(K) ∪ Range(Sp,Ep)(K);
11:         return T_CE(K)
```

The big difference between Eager and Lazy approach: After model generation for Theory solver the SMT solve function is not called.

The theory model is defined but is not solved.

Mode 3: Lazy SMT Attack

Algorithm Overall SMT Attack (*Lazy* Approach)

```
1:  function SMT_LAZY_ATT(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                          ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_CE(K) ← GenTCE(G(X, K));          ▷ Theory Constraint Expressions (Not Solved)
10:     T_CE(K1,K2) ← T_CE(K1) ∪ T_CE(K2) ;
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SATC, T_CE(K1,K2)))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC,T_CE(K));
```

Initialization of constraints for SMT attack (*Lazy* Approach)

```
1:      function GENTCE(Graph G)
2:          Inputs ← G.find_start_points();
3:          Outputs ← G.find_end_points();
4:          T_CE(K) ← []
5:          for each (Sp in Inputs) do
6:              for each (Ep in Outputs) do
7:                  Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:                  Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:                  Range(Sp,Ep)(K) ← Lower(Sp,Ep)(K) ∧ Upper(Sp,Ep)(K);
10:                 T_CE(K) ← T_CE(K) ∪ Range(Sp,Ep)(K);
11:         return T_CE(K)
```

The SMT solve function is then called to find the assignment for keys which can **satisfy both SAT solver and Theory solver(s).**

Mode 3: Lazy SMT Attack

Algorithm Overall SMT Attack (*Lazy* Approach)

```
1:  function SMT_LAZY_ATT(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      KPC ← ReplaceKPG(N_obf);
3:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
4:      KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
5:      SCKVC = TRUE;
6:      SATC = KDC ∧ SCKVC;
7:      LC = TRUE;                                          ▷ Learned Clauses
8:      G(X,K) ← Graph_Translation(N_obf);
9:      T_CE(K) ← GenTCE(G(X, K));          ▷ Theory Constraint Expressions (Not Solved)
10:     T_CE(K1, K2) ← T_CE(K1) ∪ T_CE(K2) ;
11:     while (((X_DI,K1,K2,CC) ← SMT.Solve(SATC, T_CE(K1, K2)))= TRUE) do
12:         Y_f ← C_org(X_DI);
13:         DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
14:         SCKVC = SCKVC ∧ DIVC;
15:         LC = LC ∧ CC
16:         SMT_LC = KDC ∧ SCKVC ∧ LC;
17:     Key ← SMT.Solve(SMT_LC,T_CE(K));
```

Initialization of constraints for SMT attack (*Lazy* Approach)

```
1:      function GENTCE(Graph G)
2:          Inputs ← G.find_start_points();
3:          Outputs ← G.find_end_points();
4:          T_CE(K) ← []
5:          for each (Sp in Inputs) do
6:              for each (Ep in Outputs) do
7:                  Upper(Sp,Ep)(K) ← !(distance_leq(Sp, Ep, t_cd));
8:                  Lower(Sp,Ep)(K) ← distance_leq(Sp, Ep, t_p);
9:                  Range(Sp,Ep)(K) ← Lower(Sp,Ep)(K) ∧ Upper(Sp,Ep)(K);
10:                 T_CE(K) ← T_CE(K) ∪ Range(Sp,Ep)(K);
11:         return T_CE(K)
```

The decision tree and search Space for the SMT solver is Significantly Reduced.

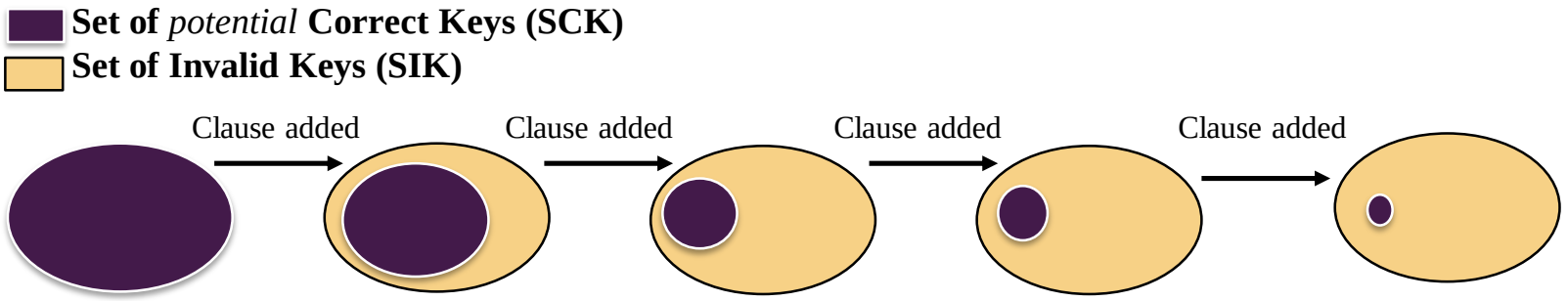
Attack Modes

- **Mode 1**: SMT **reduced** to SAT Attack
  - To show SMT is a superset of SAT
- **Mode 2**: **Eager** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Serialized!
- **Mode 3**: **Lazy** SMT Attack
  - To show the Strength of SMT
  - Theory solver(s) and SAT solver are Parallelized!
- **Mode 4**: **Accelerated** Lazy SMT Attack (AccSMT)
  - To show more efficiency
  - Uses BitVector Theory Solver

Mode 4: Accelerated SMT Attack

- DIPs are Important
  - Number of DIPs = Number of Iterations

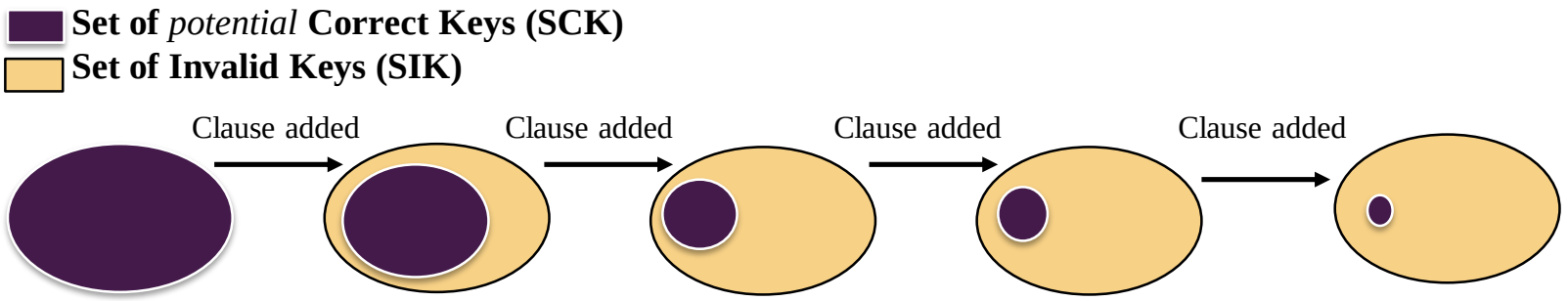
- Categorizing DIPs based on their Pruning Power
  - Stronger DIP rule outs more incorrect keys
    - Based on the number of inconsistencies that could sensitize to the primary outputs



Mode 4: Accelerated SMT Attack

- Depending of the pruning power of DIPs
  - The size of the **complete set of DIPs** could be different

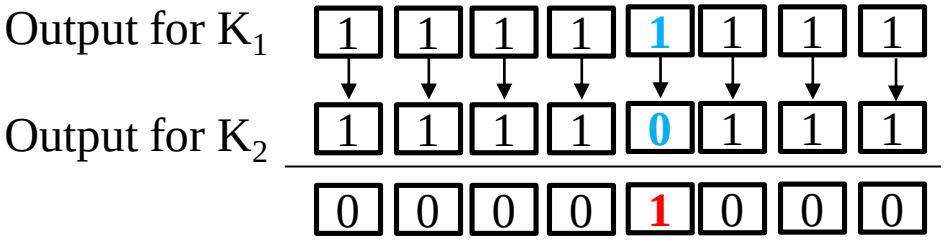
- **Minimal complete set of DIPs**
  - **The smallest set of DIPs** that could **de-obfuscate** the circuit
    - **Minimum** Number of **Iterations**
      - The Fastest Solution for De-obfuscation



Mode 4: Accelerated SMT Attack

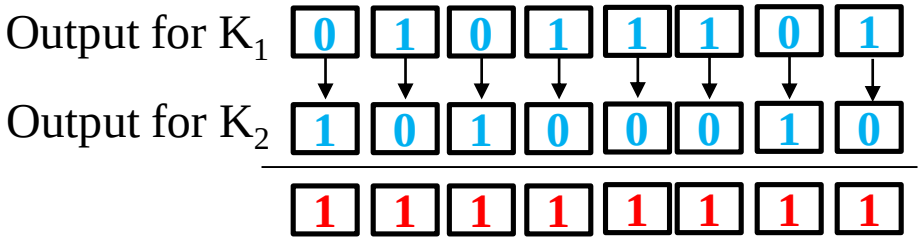
- **Lazy** approach
  - **Reduce** the size of complete set of DIPs
  - Results in **smaller** number of **iterations**

- In SAT attack only a **single difference** in the output results in generation of a DIP



Mode 4: Accelerated SMT Attack

- **Stronger requirement** for the generation of DIPs
- DIPs with the **largest Hamming Distance** in their propagated value to the primary output
- Such a DIP has a much **higher pruning capability**



Mode 4: Accelerated SMT Attack

- Assessing DIPs based on **HD of the primary output**

- Using a **BitVector** theory solver
  - Allows us to perform integer-oriented arithmetic operations
    - Addition
    - Subtraction
    - Multiplication

- The HD of output Y_1 and Y_2 is obtained using

$$HD(C(X_{DI}, K_1), C(X_{DI}, K2)) = HD(Y_1, Y_2) = \sum_{i=1}^{N} Y_1(i) \oplus Y_2(i)$$

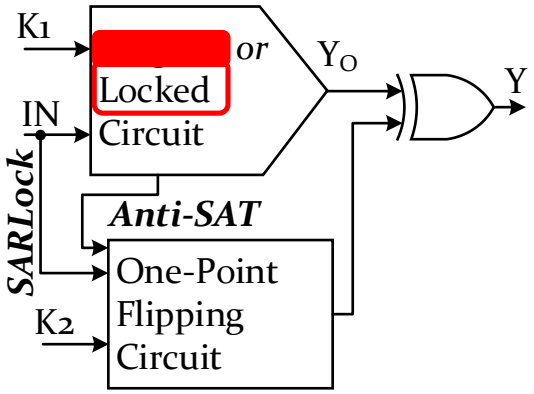
Sweeping from Maximum to Minimum (Variable)

Maximum Possible (Constant)

$$Th_{Lower} \leq HD(Y_1, Y_2) \leq Th_{Upper} = Size(Output)$$

Enabling Approximate Attack

- Applicable to **SAT-hard** Logic Locking
  - e.g. **Point functions** such as SARLock and Anti-SAT



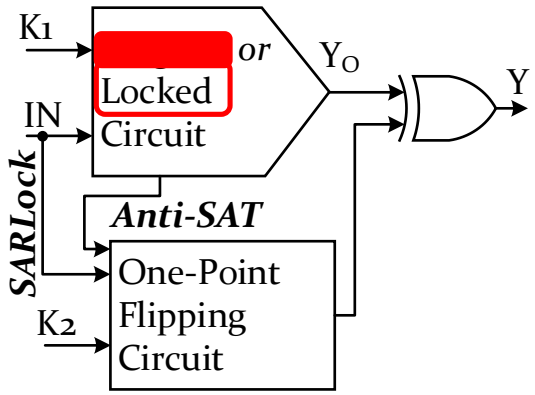
- **Point Function** obfuscation **properties**:
  - **Small output corruption**
  - Each DIP eliminates a single key value
  - The number of iterations are **exponentially large**

- To **increase the corruption** SAT hard obfuscation is usually **combined** with a high corruption obfuscation

Y_O	IN	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
✓	0	✓	✓	✓	✓	✗	✓	✓	✓
✓	1	✗	✓	✓	✓	✓	✓	✓	✓
✓	2	✓	✓	✗	✓	✓	✓	✓	✓
✓	3	✓	✓	✓	✓	✓	✗	✓	✓
✓	4	✓	✓	✓	✓	✓	✓	✓	✓
✓	5	✓	✓	✓	✓	✓	✓	✗	✓
✓	6	✓	✓	✓	✓	✓	✓	✓	✗
✓	7	✓	✓	✓	✗	✓	✓	✓	✓

Enabling Approximate Attack

- Adding a **termination strategy** in Accelerated SMT
- Using **BitVector** (based on Hamming Distance)
  - Find keys related to **high corruption obfuscation**
    - Hamming Distance > 1
  - Stop when we keep finding many keys with HD=1
    - This is the sat-hard trap zone
  - Return the Key as Approximate (with known HD)



Y_O	IN	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
✓	0	✓	✓	✓	✓	✗	✓	✓	✓
✓	1	✗	✓	✓	✓	✓	✓	✓	✓
✓	2	✓	✓	✗	✓	✓	✓	✓	✓
✓	3	✓	✓	✓	✓	✓	✗	✓	✓
✓	4	✓	✓	✓	✓	✓	✓	✓	✓
✓	5	✓	✓	✓	✓	✓	✓	✗	✓
✓	6	✓	✓	✓	✓	✓	✓	✓	✗
✓	7	✓	✓	✓	✗	✓	✓	✓	✓

Mode 4: AccSMT

Algorithm Accelerated SMT Attack

```
1:  function AccSMT_Attack(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      HD_High = Number of output bits;                    ▷ Upper hamming distance limit;
3:      HD_Low = HD_High - 1;                               ▷ Lower hamming distance limit;
4:      TO = 50s;                                           ▷ Timeout constraint;
5:      R = 20;                                             ▷ Repetition limit;
6:      R_HD = 1;                                           ▷ Repetition condition;
7:      R_count = 0;                                        ▷ Repetition count variable;
8:      KPC ← Replace_KPG(N_obf);
9:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
10:     KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
11:     SCKVC = TRUE;
12:     SATC = KDC ∧ SCKVC;
13:     LC = TRUE;                                          ▷ Learned Clauses
14:     BV(X,K) ← Circuit_Output_to_BitVector(N_obf);
15:     BVS(X, K1, K2) = SUM_of_1s(BV(X,K1) ⊕ BV(X,K2))
16:     T_CE ← BVS(X,K1,K2) ≥ HD_Low;                       ▷ Theory constraint expression;
17:     T_CE ← T_CE ∪ (BVS(X, K1, K2) ≤ HD_High);
18:     while HD_Low ≥ 1 do
19:         while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC, T_CE, TO)) = T) do
20:             Y_f ← C_org(X_DI);
21:             DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
22:             SCKVC = SCKVC ∧ DIVC;
23:             LC = LC ∧ CC
24:             SMT_LC = KDC ∧ SCKVC ∧ LC;
25:             if (HD_Low ≤ HD_R) then
26:                 if (R_count == R) then
27:                     Break;
28:                 R_count ++;
29:         HD_Low--;
30:     Key ← SMT.Solve(SMT_LC);
```

Calculate HD using BitVector

Mode 4: AccSMT

Algorithm Accelerated SMT Attack

```
1:  function AccSMT_Attack(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      HD_High = Number of output bits;                    ▷ Upper hamming distance limit;
3:      HD_Low = HD_High - 1;                               ▷ Lower hamming distance limit;
4:      TO = 50s;                                           ▷ Timeout constraint;
5:      R = 20;                                             ▷ Repetition limit;
6:      R_HD = 1;                                           ▷ Repetition condition;
7:      R_count = 0;                                        ▷ Repetition count variable;
8:      KPC ← Replace_KPG(N_obf);
9:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
10:     KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
11:     SCKVC = TRUE;
12:     SATC = KDC ∧ SCKVC;
13:     LC = TRUE;                                          ▷ Learned Clauses
14:     BV(X,K) ← Circuit_Output_to_BitVector(N_obf);
15:     BVS(X, K1, K2) = SUM_of_1s(BV(X,K1) ⊕ BV(X,K2))
16:     T_CE ← BVS(X,K1,K2) ≥ HD_Low;                       ▷ Theory constraint expression;
17:     T_CE ← T_CE ∪ (BVS(X, K1, K2) ≤ HD_High);
18:     while HD_Low ≥ 1 do
19:         while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC, T_CE, TO)) = T) do
20:             Y_f ← C_org(X_DI);
21:             DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
22:             SCKVC = SCKVC ∧ DIVC;
23:             LC = LC ∧ CC
24:             SMT_LC = KDC ∧ SCKVC ∧ LC;
25:             if (HD_Low ≤ HD_R) then
26:                 if (R_count == R) then
27:                     Break;
28:                 R_count ++;
29:         HD_Low--;
30:     Key ← SMT.Solve(SMT_LC);
```

Maximum HD = Output Width (Constant)

Minimum HD = Starts from Maximum

Minimum HD ➔ Sweeping (Iteratively Decreasing)

Experimental Results

- Evaluation of SMT reduced to SAT Attack
  - **Purpose:** to show SMT is **superset** of SAT
  - We experimented using **two obfuscation** methods
    - random XOR/XNOR insertion (**RLL**)
    - obfuscation using nets with unbalanced probabilities **IOLTS'14**
  - We used **ISCAS-85 benchmarks** with obfuscation overhead ranging from 1% to 25%.

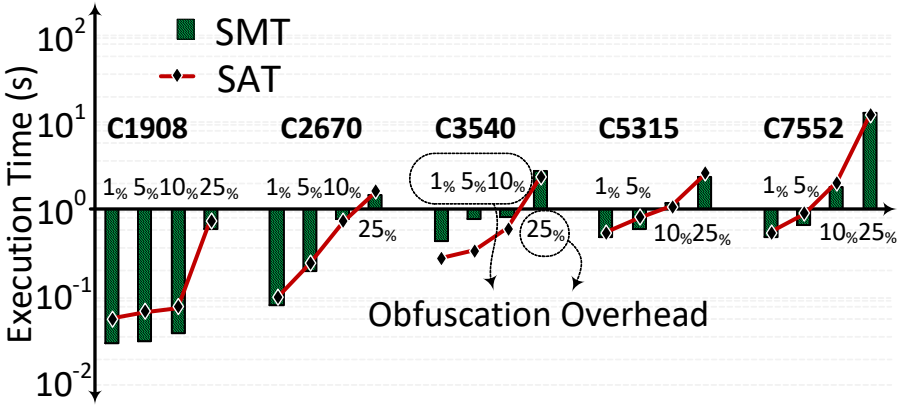
Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c7552
# of Inputs	36	41	60	41	33	233	50	178	207
# of Outputs	7	32	26	32	25	140	22	123	108
# of Gates	120	162	320	506	603	872	1179	1726	2636

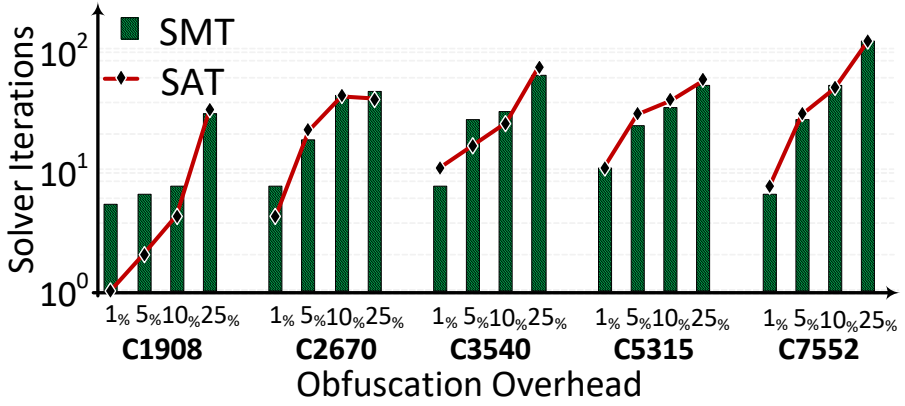
Evaluation of SMT Reduced to SAT

- Random XOR/XNOR insertion (RLL)

Circuit	c2670				c3540				c5315				c7552			
	SAT		SMT		SAT		SMT		SAT		SMT		SAT		SMT	
	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time
1%	3	0.102	5	0.474	10	0.513	8	1.31	9	0.405	10	0.441	11	0.577	19	0.806
5%	45	1.514	57	3.589	19	1.502	25	1.249	32	1.354	24	2.433	67	5.271	42	4.261
10%	312	14.08	342	15.752	36	1.782	36	2.973	59	3.798	57	4.881	97	15.82	94	15.67
25%	781	114.5	692	108.6	77	9.796	65	8.462	95	19.63	107	22.48	215	225.6	228	270.8

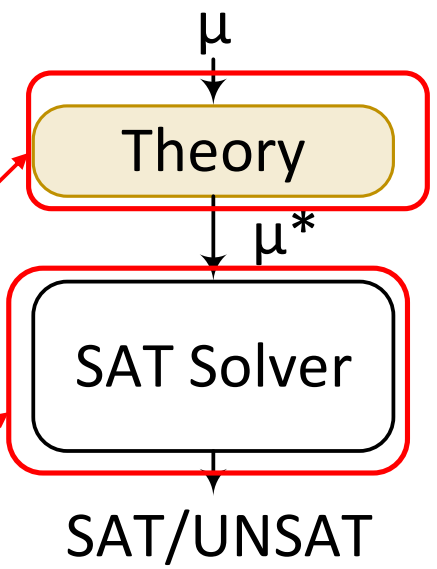
- Obfuscation using nets with unbalanced probabilities IOLTS'14





Evaluation of Eager SMT Attack

- **DLL as the case study**
  - cannot be modeled in a SAT attack.
  - DLL + MUX/XOR-based logic locking
- **Serial** invocation of theory and SAT solver.

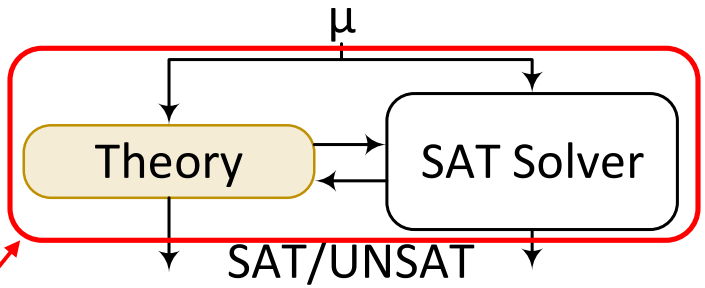


Circuit	c1908	c2670	c3540	c5315	c7552
1%	0.077 + 1.663	0.068 + 170.0	0.053 + 4.054	1.291 + 114.6	0.580 + 138.6
2%	0.016 + 1.919	0.221 + 175.6	0.200 + 5.001	1.535 + 144.6	1.808 + 185.5
3%	0.054 + 2.161	0.337 + 212.7	1.359 + 6.328	3.057 + 160.4	2.247 + 245.9
5%	0.075 + 2.810	0.495 + 248.4	1.553 + 8.325	3.891 + 256.9	7.812 + 353.3
10%	0.499 + 3.812	38.78 + 407.1	1.524 + 14.35	16.19 + 550.3	33.92 + 782.7
25%	8.951 + 21.71	112.4 + 972.5	9.459 + 92.42	60.30 + 1567	2920 + 5244

SMT execution time = $x + y$, x : The execution time of the SAT engine of the SMT Solver,
y : The execution time of the theory engine of the SMT Solver

Evaluation of Lazy SMT Attack

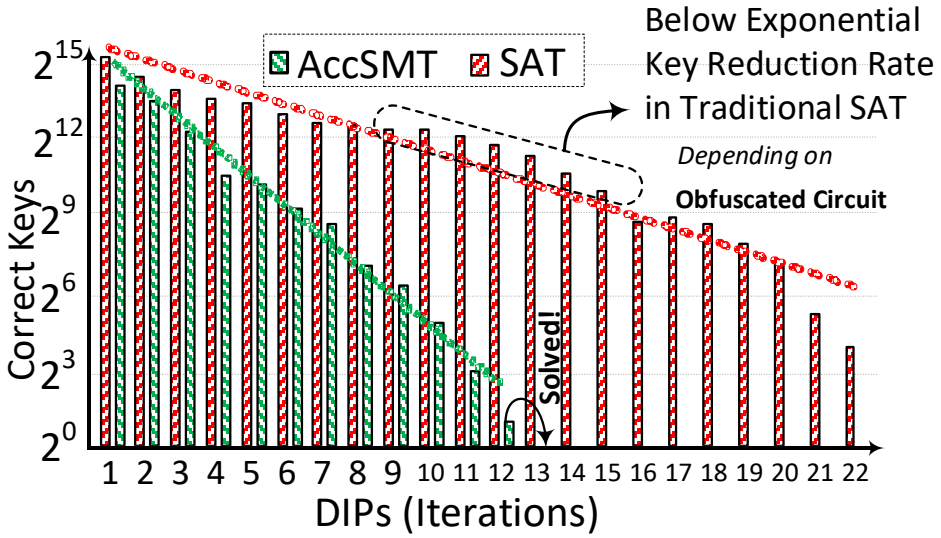
- **DLL as the case study**
  - cannot be modeled in a SAT attack.
  - DLL + MUX/XOR-based logic locking
- **Parallel** invocation of theory and SAT solver

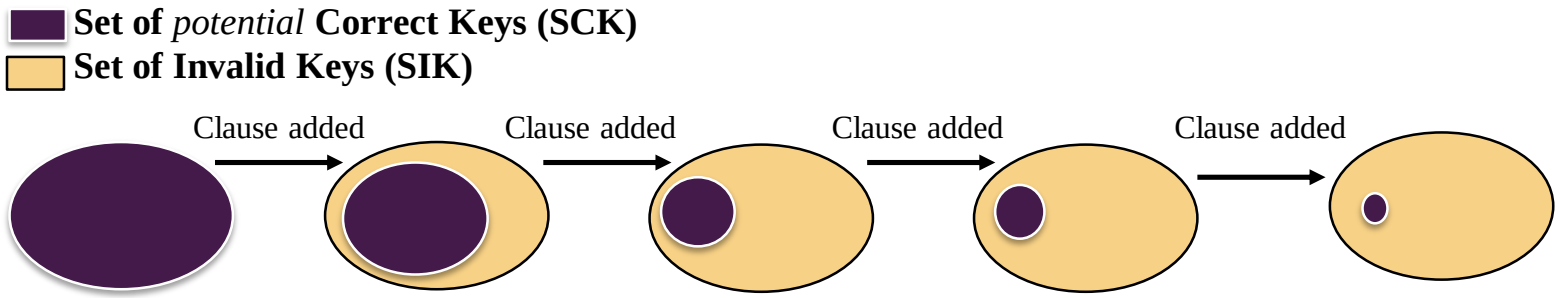


Circuit	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c7552
1%	0.033	0.177	0.263	0.567	0.466	20.44	0.983	11.53	13.07
2%	0.049	0.262	0.325	0.676	0.596	21.86	3.443	11.76	17.83
3%	0.065	0.329	0.350	0.877	0.723	23.39	2.436	15.27	19.04
5%	0.049	0.340	0.517	1.085	1.456	28.87	2.587	38.87	45.96
10%	0.204	0.503	1.195	5.622	3.334	83.06	6.712	94.80	319.6
25%	0.599	1.481	2.036	297.2	95.67	2706	126.3	552.8	8045

Evaluation of AccSMT

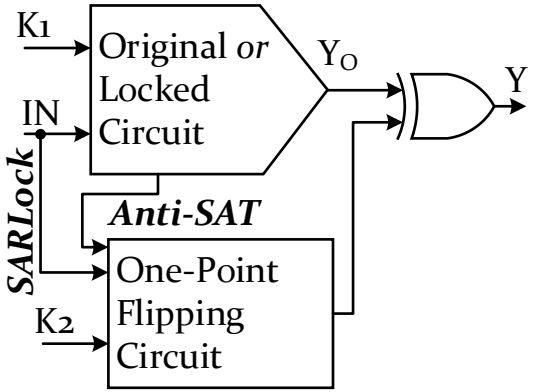
- Ability to find **stronger DIPs**
  - **Pruning power** of DIPs is higher!
  - Higher rate in decreasing the number of remaining keys





Enabling Approximate Attack

- SARLock + IOLTS'14
  - Finds the **correct keys** for **high-corruption**
  - **detects** the SAT-hard **trap**
  - **exits**, and reports the **approximate key**



Circuit	c1908		c2670		c3540		c5315		c7552	
	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time
1%	7	0.512	16	3.075	8	1.304	3	0.384	7	2.905
5%	18	0.701	25	11.91	15	1.681	11	1.707	33	17.56
10%	31	4.085	51	26.47	21	3.779	35	7.402	61	44.07
25%	71	8.605	105	76.8	66	22.91	56	16.64	88	58.32

Conclusion

- **introduced** the powerful **SMT attacks**
  - Benefits from the expressive nature of **theory solvers**
- Proved that SMT attack is **a superset of the SAT** attack
- Explained the **Eager** and **Lazy** mode of SMT attack
- Using both Eager and Lazy approach
  - We **broke the DLL** obfuscation That cannot be broken by a SAT attack
  - Why? SMT attack's capabilities go beyond a SAT attack
- Presented the **accelerated SMT** attack (AccSMT)
  - significant **speed-up** compare to pure SAT attack
- Presented a formulation for SMT **approximate attack**
  - To find an approximate key for compound obfuscation schemes

Selected References

[1] J. Roy *et al*. 2010. Ending piracy of integrated circuits. Computer, 43, 10 (2010), 30–38.

[2] P. Tuyls *et al*. 2006. Read-proof hardware from protective coatings. In CHES. 369–383.

[3] J. Rajendran *et al*. 2012. Security analysis of logic obfuscation. In DAC. 83–89.

[4] K. Shamsi *et al*. 2017. AppSAT: Approximately deobfuscating integrated circuits. In HOST. 95–100.

[5] M. Yasin *et al*. 2017. Removal attacks on logic locking and camouflaging techniques. IEEE Trans. on Emerging Topics in Computing1 (2017).

[6] P. Subramanyan *et al*. 2015. Evaluating the security of logic encryption algorithms. In HOST.137–143.

[7] Y. Shen and H. Zhou. 2017. Double dip: Re-evaluating security of logic encryption algorithms. In GLSVLSI. 179–184.

[8] D. Sirone and P. Subramanyan. 2018. Functional Analysis Attacks on Logic Locking. arXiv preprint arXiv:1811.12088 (2018).

[9] M. Yasin *et al*. 2016. SARLock: SAT Attack Resistant Logic Locking. In HOST. 236–241.

[10] Y. Xie and A. Srivastava. 2016. Mitigating sat attack on logic locking. In CHES. 127–146.

[11] M. Yasin *et al*. 2017. Provably-secure logic locking: From theory to practice. In ACM-CCS. 1601–1618.

[12] H. M. Kamali *et al*. 2019. Full-Lock: Hard Distributions of SAT Instances for Obfuscating Circuits using Fully Configurable Logic and Routing Blocks. In DAC. 6.

[13] S. Roshanisefat *et al*. 2018. SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware. In GLSVLSI. 153–158.

[14] Y. Xie *et al*. 2017. Delay locking: Security enhancement of logic locking against ic counterfeiting and overproduction. In DAC. 9.

[15] M. Yasin *et al*. 2017. Security analysis of anti-sat. In ASP-DAC. 342–347.

[16] H. Zhou *et al*. 2017. CycSAT: SAT-based attack on cyclic logic encryptions. In ICCAD. 49–56.

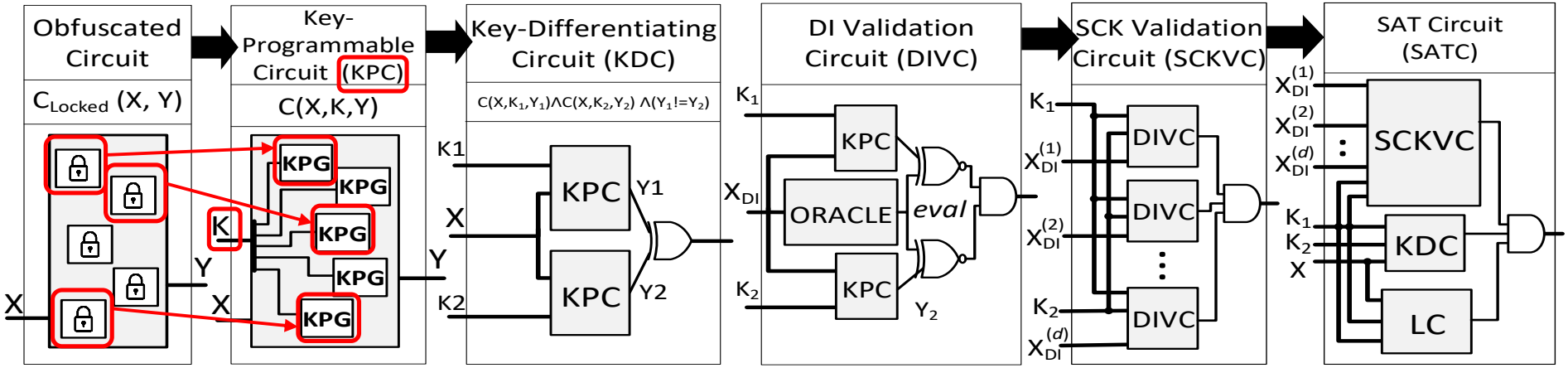
[17] Y. Shen *et al*. 2019. BeSAT: behavioral SAT-based attack on cyclic logic encryption. In ASP-DAC.ACM, 657–662.

[18] C. Barrett *et al*. 2015. Satisfiability modulo theories. In Handbook of Model Checking. 305–343.

[19] S. Bayless *et al*. 2015. Sat Modulo Monotonic Theories. In AAAI. 2015. 3702–3709.

SAT Attack

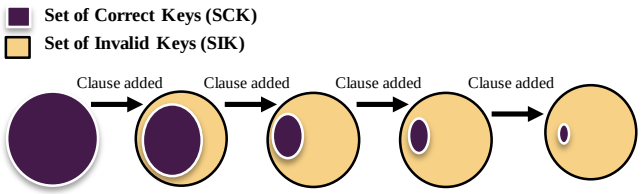
- SAT Attack



Replace all obfuscated cells
with key programmable gates

+

Adding Key Inputs



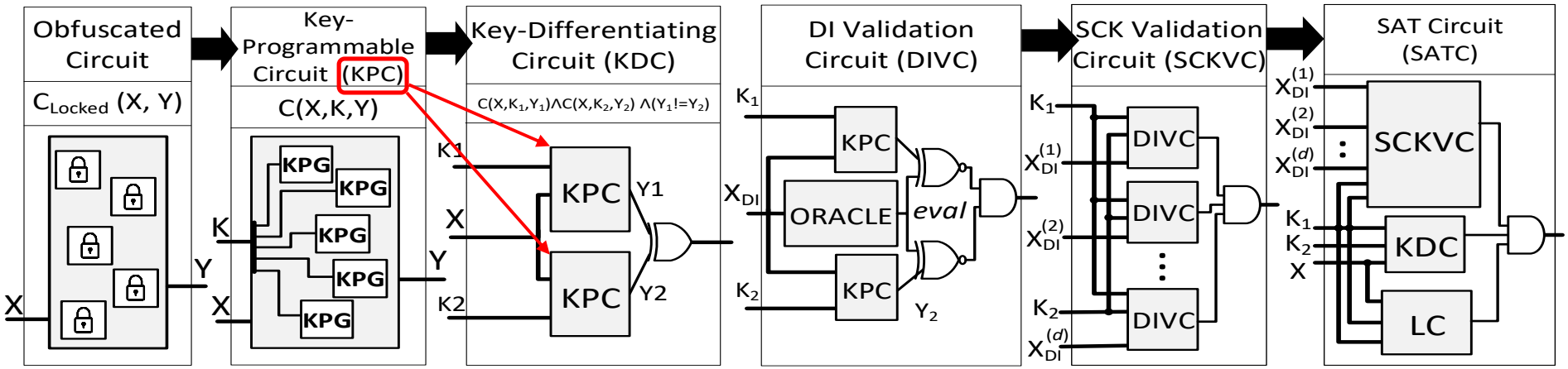
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

```
1: function SAT_ATTACK(Circuit C_L, Circuit C_O)
2:     i ← 0; F_0 ← C_L(X, K_1, Y_1) ∧ C_L(X, K_2, Y_2);
3:     while SAT(F_i ∧ (Y_1 ≠ Y_2)) do
4:         X_d[i] ← sat_assignment (F_i ∧ (Y_1 ≠ Y_2)); Y_d[i] ← C_O(X_d[i]);
5:         F_{i+1} ← F_i ∧ C_L(X_d[i], K_1, Y_d[i]) ∧ C_L(X_d[i], K_2, Y_d[i]); i ← i+1;
6:     K* ← sat_assignment_{K_1}(F_i);
```

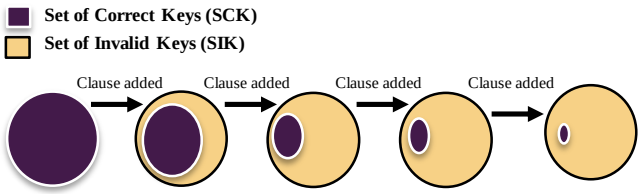
SAT Attack

- SAT Attack



Duplicating KPC

- Primary Inputs are in Common
- Keys are Different
- XORed



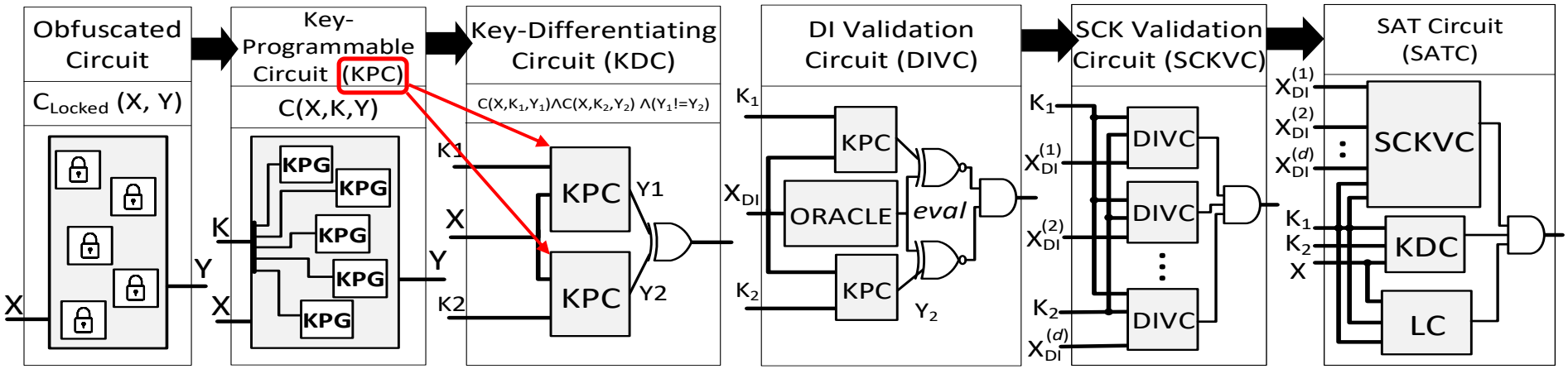
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

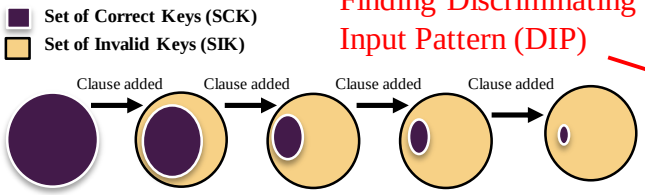
1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

SAT Attack

- SAT Attack



Duplicating KPC
- Primary Inputs are in Common
- Keys are Different
- XORed



Breaks within few minutes (few iterations)!

Finding Discriminating Input Pattern (DIP)

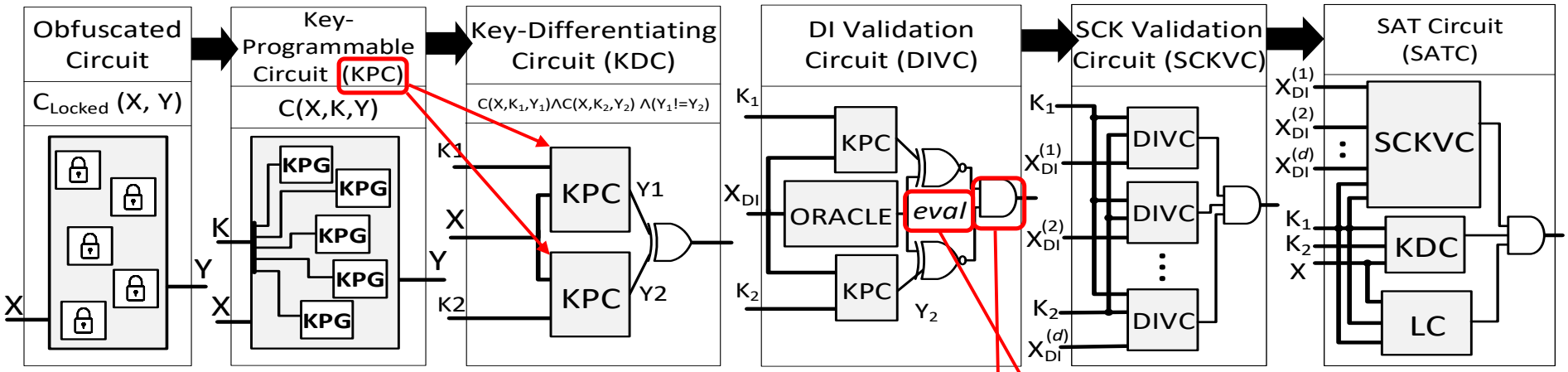
Algorithm SAT-based Attack Algorithm

1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

Observing C_O for new DIP

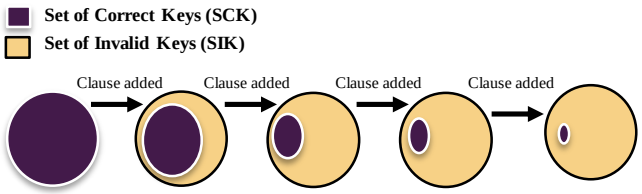
SAT Attack

- SAT Attack



Validating the found DIP

- DI validation Circuit confirms that two new keys produce the same correct output for a previously found DI



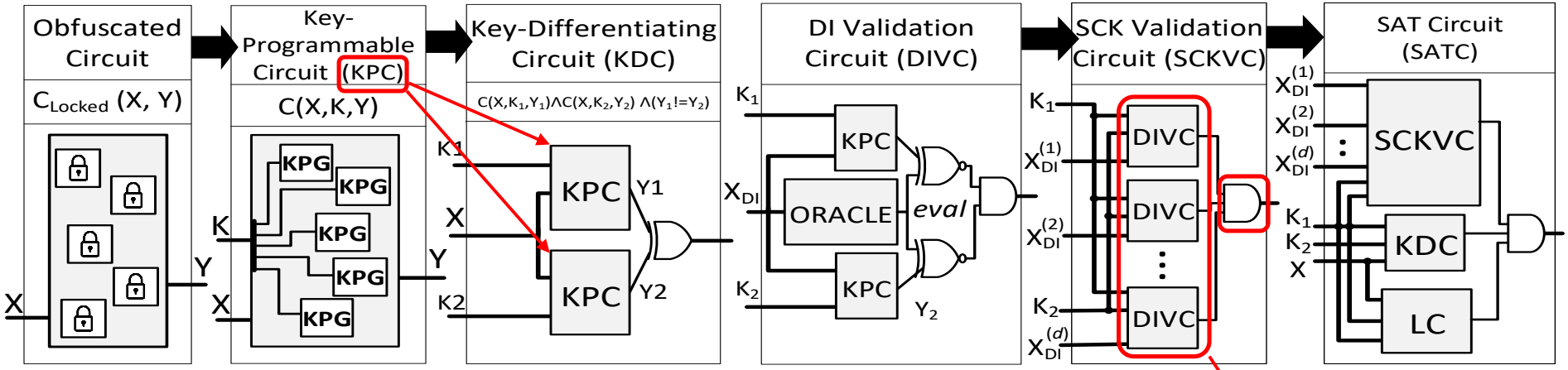
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

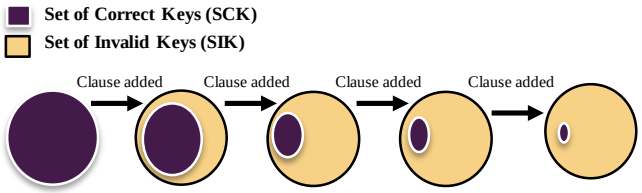
1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

SAT Attack

- SAT Attack



Iteratively, Finding and Validating new DIP
- ANDed to confirms all of the previously found DIPs



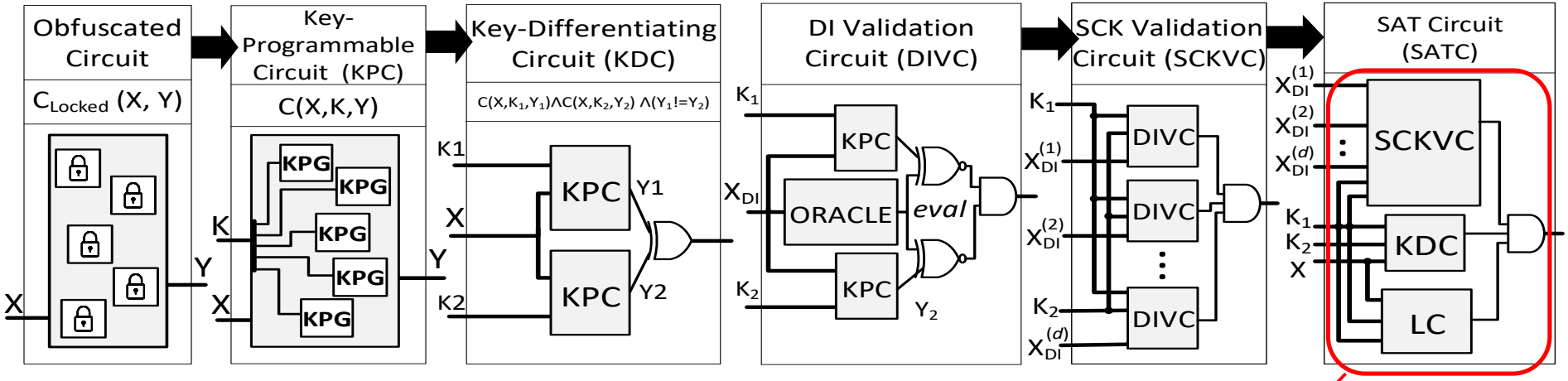
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

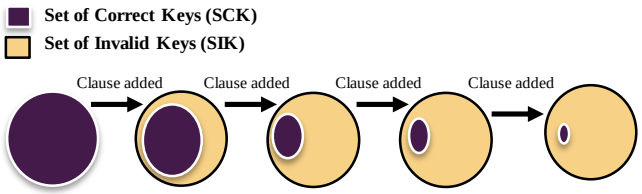
SAT Attack

- SAT Attack



Terminate Condition:

- SAT is not able to find a new DIP
- Finding Correct Key based on all previously found DIPs.



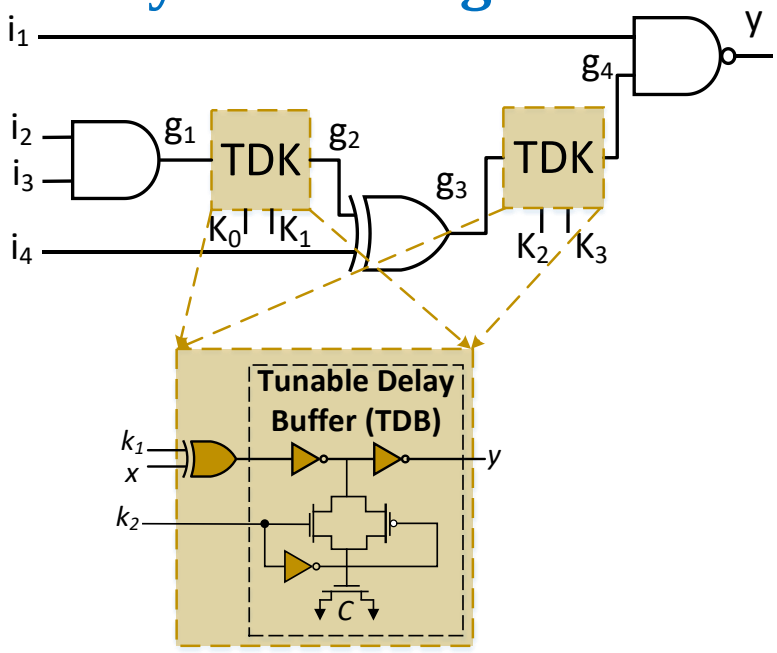
Breaks within few minutes (few iterations)!

Algorithm SAT-based Attack Algorithm

1: **function** SAT_ATTACK(Circuit C_L, Circuit C_O)
2: $i \leftarrow 0$; $F_0 \leftarrow C_L(X, K_1, Y_1) \wedge C_L(X, K_2, Y_2)$;
3: **while** $SAT(F_i \wedge (Y_1 \neq Y_2))$ **do**
4: $X_d[i] \leftarrow$ sat_assignment $(F_i \wedge (Y_1 \neq Y_2))$; $Y_d[i] \leftarrow C_O(X_d[i])$;
5: $F_{i+1} \leftarrow F_i \wedge C_L(X_d[i], K_1, Y_d[i]) \wedge C_L(X_d[i], K_2, Y_d[i])$; $i \leftarrow i+1$;
6: $K^* \leftarrow$ sat_assignment$_{K_1}(F_i)$;

Mode 2: Eager SMT Attack

- By having *K* TDK cells
  - 2*K* in total
  - SAT solver returns one logically correct key sequence among (2^k) different set
    - Only **one of such key** does not result in setup and hold time violations
- The correct attack should **consider the delay and timing properties** of the netlist
  - In addition to its logical correctness!



Using TimeOut

- The execution of SAT and SMT attack

$$\sum_{i=1}^{N} t(i)$$

t(i) is the execution time of the ith iteration of an SMT attack

- By just reducing the number of SAT iterations (*N*)
  - We cannot guarantee a shorter execution time
  - We limit the time allowance for finding a DIP in each iteration

- *TO* prevents the SMT solver from spending a long time for finding a DIP with large HD

Using TimeOut(Cont.)

- *TO* prevents the SMT solver from spending a long time for finding a DIP with large HD

- By adapting TO feature during SMT attack, the HD requirement is reduced
  - The SMT solver returns UNSAT
    - There is no such input
  - We encounter TO interrupt
    - The HD constraints posed on BitVector theory solver is reduced by one
    - The SMT solver is called

- Time interrupt is supported by MonoSAT used in this paper
- Our experiments illustrate that this usually results in considerably smaller execution time.

Mode 4: AccSMT

Algorithm Accelerated SMT Attack

```
1:  function AccSMT_Attack(Obfuscated_Netlist N_obf, Functional_Circuit C_org)
2:      HD_High = Number of output bits;                    ▷ Upper hamming distance limit;
3:      HD_Low = HD_High - 1;                               ▷ Lower hamming distance limit;
4:      TO = 50s;                                           ▷ Timeout constraint;
5:      R = 20;                                             ▷ Repetition limit;
6:      R_HD = 1;                                           ▷ Repetition condition;
7:      R_count = 0;                                        ▷ Repetition count variable;
8:      KPC ← Replace_KPG(N_obf);
9:      C(X,K,Y) ← Circuit_Translation_to_CNF(KPC);
10:     KDC = C(X, K1, Y1) ∧ C(X, K2, Y2) ∧ (Y1 ≠ Y2);
11:     SCKVC = TRUE;
12:     SATC = KDC ∧ SCKVC;
13:     LC = TRUE;                                          ▷ Learned Clauses
14:     BV(X,K) ← Circuit_Output_to_BitVector(N_obf);
15:     BVS(X, K1, K2) = SUM_of_1s(BV(X,K1) ⊕ BV(X,K2))
16:     T_CE ← BVS(X,K1,K2) ≥ HD_Low;                       ▷ Theory constraint expression;
17:     T_CE ← T_CE ∪ (BVS(X, K1, K2) ≤ HD_High);
18:     while HD_Low ≥ 1 do
19:         while (((X_DI,K1,K2,CC) ← SMT.Solve(SMT_LC, T_CE, TO)) = T) do
20:             Y_f ← C_org(X_DI);
21:             DIVC = C(X_DI,K1,Y_f) ∧ C(X_DI,K2,Y_f);
22:             SCKVC = SCKVC ∧ DIVC;
23:             LC = LC ∧ CC
24:             SMT_LC = KDC ∧ SCKVC ∧ LC;
25:             if (HD_Low < HD_R) then
26:                 if (R_count == R) then
27:                     Break;
28:                 R_count ++;
29:         HD_Low--;
30:     Key ← SMT.Solve(SMT_LC);
```

Check Repetitive ➔ avoid trapping (Approximate Key)

Evaluation of AccSMT

- Comparison of the execution time and the number of iterations between the SAT solver and the AccSMT solver
- AccSMT attack is carried in a smaller number of iterations and requires order(s) of magnitude smaller execution time

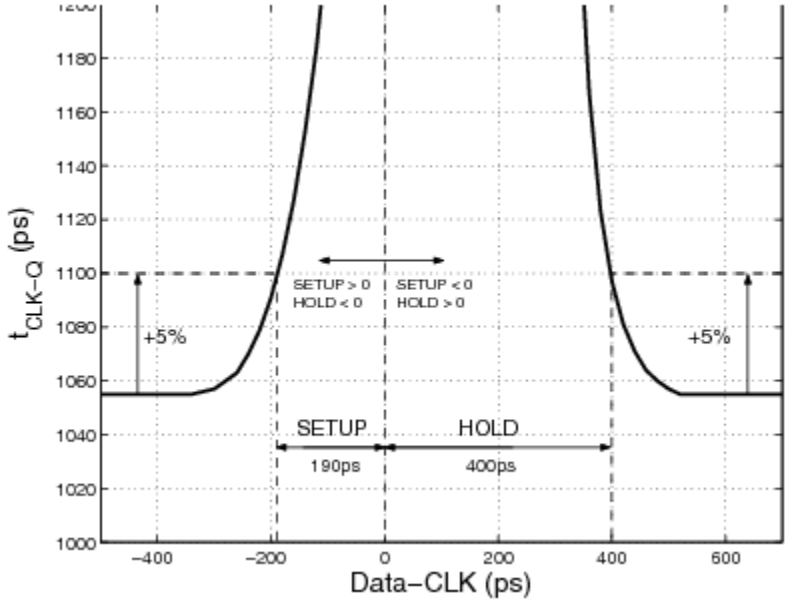
Circuit	c2670				c3540				c5315				c7552			
	SAT		AccSMT		SAT		AccSMT		SAT		AccSMT		SAT		AccSMT	
	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time
1%	3	0.102	2	0.316	10	0.513	3	0.185	9	0.405	2	0.163	11	0.577	3	0.374
5%	45	1.514	11	3.589	19	1.502	6	0.761	32	1.354	6	0.408	67	5.271	17	2.607
10%	312	14.08	26	5.817	36	1.782	11	1.236	59	3.798	12	1.753	97	15.82	19	4.721
25%	781	114.5	107	24.05	77	9.796	16	1.606	95	19.63	27	7.916	215	225.6	24	23.52

Eager vs. Lazy

- Eager vs. Lazy
  - **majority of cases**:
    - Lazy approach outperforms the Eager
  - **some cases** (e.g. for Benchmark C1908 with 50% overhead):
    - The Lazy approach is slower than Eager approach
    - Lazy approach doesn't always result in the stronger attack
  - There exist **a set** of problems that **Eager is not even applicable**
    - leaving the Lazy approach as the only solution forward

Calculating Setup time and Hold time

- Standard flow for extracting the register setup time
  - we are looking at 5% increase in $t_{clk\text{-}q}$ when we sweep the data-clock arrival time difference from a large negative to a large positive number.



Conjunctive Normal Form (CNF)

- A Conjunction of one or more Clauses
  - each Clause is a Disjunction of Literals

- Similar to Product of Sum (PoS)

  - C = not(A) ➔ $(\overline{A} \vee \overline{C}) \wedge (A \vee C)$