HOST: HLS Obfuscations against SMT ATtack

Chandan Karfa* and TM Abdul Khader* and Yom Nigam* and Ramanuj Chouksey* and Ramesh Karri†
*Indian Institute of Technology Guwahati, India
†New York University, USA
*ckarfa@iitg.ac.in, *{abdulkhader.magnetta, yom, r.chouksey}@alumni.iitg.ac.in †rkarri@nyu.edu

***Abstract*—The *fab-less* IC design industry is at risk of IC counterfeiting and Intellectual Property (IP) theft by untrusted third party foundries. Logic obfuscation thwarts IP theft by locking gate-level netlists using a locking key. The complexity of circuit designs and migration to high level synthesis (HLS) expands the scope of locking to a higher abstraction. Automated RTL locking during HLS integrates obfuscation into the back-end HLS tool. This is tedious and requires access to the HLS tool source code. Furthermore, recent work proposed an SMT attack on HLS-based obfuscation. In this work, we propose sn RTL locking tool HOST, to thwart the SMT attack. The HOST approach is agnostic to the HLS tool. Results show that HOST obfuscations have low overhead and thwart SMT attacks.**

***Index Terms*—Register Transfer Level, High Level Synthesis, Logic Locking, RTL Locking**

I. Introduction

Globalization of the supply chain poses security threats. Since the foundry has access to the design layout, rogue employees may misuse the layout to create illegal copies of the ICs, undermining the legitimate market of the design house. This forces the fab-less design houses to protect their IP. IC counterfeiting and IP theft cost millions of dollars every year to the semiconductor industries.

Logic locking has been proposed to resist IP theft and counterfeiting [13]. Logic locking embeds circuitry into the design to lock its functionality using a key that is unknown to the foundry. Post fabrication, the key is applied to activate the IC. Until recently, hardware security literature focused on gate-level locking and attacks on locked netlists. The Algorithm-level Obfuscator (TAO) [10] locks the IC functionality at a higher abstraction during high-level synthesis (HLS).However, TAO is vulnerable to the SMT attack [9].

Also, since TAO applies obfuscations during HLS it needs access to HLS tool source. TAO has three limitations: (i) Scalability: TAO obfuscation need to be implemented in all HLS tools. (ii) Verification: When TAO obfuscation is done during HLS, the generated RTL is different from the HLS-generated RTL without obfuscation. While HLS verification is a difficult problem [4], verifying correctness of the TAO RTL against the source code is more difficult. (iii) Backdoors: The verification challenges allow a rogue in the EDA design house to insert a backdoor using a secondary key. Pre-silicon verification of the design against all possible key combinations is not viable and the secondary key can open the functional IC to the adversary. The objectives of HOST are two-fold.

1) Develop RTL obfuscations that are HLS tool agnostic.
2) Develop RTL obfuscations that are SMT-attack resilient.

The contributions of HOST are three-fold:

- HOST is HLS-independent obfuscation of the FSM and datapath in HLS-generated RTL.
- HOST RTL obfuscations thwart the SMT attack [9].
- The overhead of HOST is evaluated on HLS benchmarks.

The obfuscation techniques discussed in this paper exploits the finite state machines with datapath (FSMD) structure of HLS based RTLs. Therefore, these techniques could be applied to the RTLs generated by any HLS tool.

The paper is organized as follows: Section II describes the background work. Section III describes a set of generic RTL obfuscations supported by the HOST framework. SMT attack resilient RTL obfuscations are described in Section IV. We experimentally evaluate the HOST framework in Section V-A. The paper concludes in Section VI.

II. Background

A. Threat Model

We assume the attacker is a rogue employee in the foundry aiming at stealing the IP. As in prior works, HOST assumes the foundry can extract the gate-level netlist of the locked IC from its GDSII layout. The foundry extracts RTL descriptions of the datapath and FSM from the gate-level description of the locked IC using existing techniques [11]. Finally, the foundry recovers the locking key with SMT attacks [9]. It assumes access to a functioning (unlocked) copy of the IC from the market.The goal for HOST is to develop obfuscations for HLS generated RTL to thwart SMT-based attacks.

B. Related Works

Logic obfuscation thwarts IP theft and counterfeiting by malicious foundries [13]. A locking key obfuscates the functional access to the chip. Existing logic obfuscation tools operate on gate-level netlists [5], [12], [13]; extra gates obfuscate IC functionality which could be unlocked using the locking key, post fabrication. A recent approach implements obfuscation at the algorithmic-level where actionable semantics are available [2], [6], [8], [10], [20]. Pilato et al. [10] apply algorithmic modifications to the HLS back-end to integrate RTL obfuscation. SFLL-HLS [20] analyzes the input C code to determine large add and subtracts in a design and locks these units using SFLL using the LegUp HLS tool. In both instances, obfuscations are implemented in HLS source. [9] developed an SMT attack that recovered the locked RTL [10].

HOST exploits properties of HLS-generated RTL-finite state machine with datapaths (RTL-FSMDs) like separability of

FSM and datapath to add obfuscations into the datapath and the FSM. HOST emphasizes resiliency of the locked design against the SMT attack [9]. Since obfuscation is implemented on the RTL design, HOST is HLS tool agnostic.

C. TAO

TAO [10] is an algorithm-level obfuscation technique that obfuscates constants, control branches and datapath operations using a locking key K during HLS. The circuit will work correctly for the correct key. A constant c_i^p is locked as $c_i^e = c_i^p \oplus k_i$, where c_i^e is the locked value stored in hardware and k_i is a fixed x-bit key. The actual constant can be obtained as $c_i^p = c_i^e \oplus k_i$ when k_i is known. A branch condition $(c_p == 1)$ is locked as $(c_p \oplus k_j == 1)$, where k_j is a one bit key. The right branch will be taken for the correct k_j. An operation $a = b + c$ is locked with a key k_j as $a = k_j \ ? \ b - c : b + c$ where $b - c$ is a dummy operation. The position of the correct operation $b + c$ is either in the if or else part based on the key.

D. SMT Attack

In [9], a satisfiability modulo theories (SMT)-based algorithm can recover the secret key of TAO-locked RTL. The algorithm extracts an RTL-FSMD from an RTL design by applying the rewriting approach [3]. Since the HLS-generated RTL separates the datapath and the controller, RTL-FSMD generation is possible. The rewriting method identifies the RTL operations performed in the datapath for the control signal assertions in each state of FSM. This way the FSM is converted to RTL-FSMD and is used in the SMT attack. Similar to the SAT attack [14], the SMT attack finds distinguishing input patterns (DIPs) iteratively to rule out equivalence classes of incorrect keys and stops when no DIPs are found.

III. HOST RTL Obfuscations

TAO [10] obfuscates control branches, constants and arithmetic operations in the C source during HLS. HOST extends this to the RTL by obfuscating branches of the FSM, modifying constants, and multiplexing arithmetic units.

A. Control Branch Obfuscation

The FSM has several states where the control signals are assigned 1/0 values and control branches that direct the sequential flow of FSM based on a set of status signals from the datapath. The conditional transitions can be obfuscated using locking key bits so that the wrong keys redirect the FSM sequence to an undesirable state. Listing 1 shows an extract from a generic FSM. The FSM makes either *STATE_2→STATE_1* transition or the *STATE_2→STATE_3* transition based on the Boolean condition which is obfuscated by a key bit *key[0]*.

B. Arithmetic and Constant Obfuscations

Like TAO, HOST identifies the operations and constants in the datapath and obfuscates them. Listing 2 shows a code segment with a series of constant additions. Add and the three constants can be obfuscated using 25 key bits. HOST uses an 8-bit key to obfuscate each constant and one key bit to obfuscate the add.

```
/*************Original code snippet**************/
STATE_2 : begin
    if (Boolean_cond == 1'b1) begin
        NEXT_STATE = STATE_1;
    end else begin
        NEXT_STATE = STATE_3;
    end
end
/************Obfuscated Code snippet*************/
STATE_2 : begin
    if (obfus__cond ^ key[0]  == 1'b1) begin
        NEXT_STATE = STATE_3;
    end else begin
        NEXT_STATE = STATE_1;
    end
end
```

Listing 1: HOST Branch Obfuscation

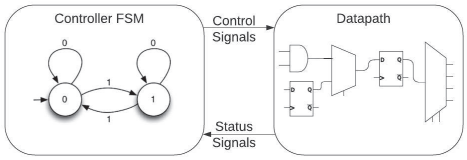

Fig. 1: FSMD structure of HLS generated RTL

IV. SMT-Attack Resilient HOST Obfuscations

Fig 1 shows the structure of a HLS generated RTL. The HLS generated RTL has an FSM, a datapath and a communication network of control and status signals. HOST manipulates the FSM+datapath (FSMD) of a HLS-generated RTL to yield SMT attack [9] resilient obfuscations. As discussed in Section II-D, the SMT attack [9] extracts the high-level behavior (i.e., RTL-FSMD) by exploiting the FSMD architecture of the generated RTL. It applies the SAT-based SMT attack on this RTL-FSMD to recover the keys. Execution time of the SMT attack depends on the number of SMT iterations (N) and the time required to solve one iteration (T_i). The SMT attack can

```
/*************Original code snippet**************/
always @(posedge clk) begin
 if (cond == 1'b0) begin
    reg_X = reg_X + const_2;
    count = count + const_3;
 end
end
assign cond = (count == const_1);
/************Obfuscated Code snippet*************/
always @(posedge clk) begin
 if (cond == 1'b0) begin
  if (key[24] == 1'b1) begin
    reg_X = reg_X + (obf_const_2 ^ key[7:0]);
  end
  else begin
    reg_X = reg_X * (obf_const_2 ^ key[7:0]);
  end
    count = count + (obf_const_3 ^ key[15:8]);
 end
end
assign cond=(count == (obf_const_1 ^ key[23:16]));
```

Listing 2: HOST arithmetic+constant obfuscation

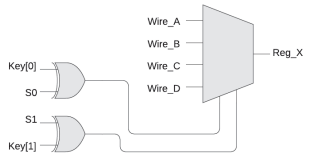

Fig. 2: HOST control signal obfuscation

be thwarted if either N or T_i is large enough to make SAT attack infeasible within a reasonable time. HOST defense uses three obfuscations: (i) Obfuscate RTL so that generating a unique abstract behaviour (RTL-FSMD) is not possible, (ii) Create difficult instances for SMT solvers (T_i very high), and (iii) $\uparrow$ iterations for SMT (N very high).

A. HOST Makes Unique Abstractions Impossible

Control signal obfuscation: In each state of the FSM, a set of control signals activate functional units and the routing network in the datapath to perform register transfer operations. HOST obfuscates these control signals to execute a false trace for an incorrect key. Listing 3 is an extract of state 4 from a sample FSM where S0 and S1 are the select inputs of a multiplexer in Fig 2. HOST obfuscates these signals with keys $key[0]$, $key[1]$, respectively. Register rewriting in [9] retraces the register transfer operations from the datapath using the control signals. When the control signals are obfuscated, the rewriting creates multiple RTL-FSMDs corresponding to each possible value of obfuscated control signals. For 'N' obfuscated control signals, 2^N RTL-FSMD equivalents are generated. Corresponding to the two control signals (S0, S1) in Fig 2 four RTL-FSMDs are generated with four distinct register assignments to *reg_X*.

```
/*************Original code snippet**************/
STATE_4: begin
     {S0,S1} = 2'd10; // Control signal transfers
     NEXT_STATE = STATE_5;
end
/************Obfuscated Code snippet*************/
STATE_4: begin
     {S0,S1} = 2'd00 ^ {key[0],key[1]}; //locked
     NEXT_STATE = STATE_5;
end
```

Listing 3: HOST control signal obfuscation

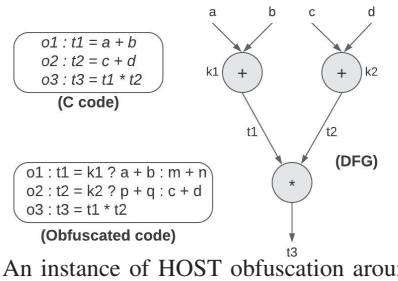

Fig. 3: An instance of HOST obfuscation around multiplier

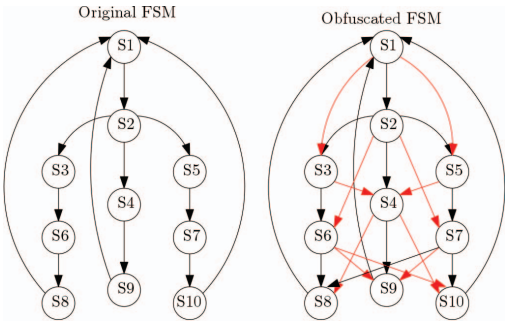

Fig. 4: FSM obfuscation using spurious transitions

B. HOST Creates Hard Instances for SMT solvers

SMT solvers do not fare well in the presence of non-linear operations. The SMT attack in [9] uses QF_AUFBV arithmetic. From our empirical studies, the Z3 solver times out for non-linear operations like *shift and multiplication with variables*. Arithmetic and constant obfuscation using such operations slows down the solver.

1) Obfuscates Constants in Shift Operations: If the constant in the left (/right) shift is obfuscated with a key, it becomes a shift by a variable amount. The symbolic simulator Klee used in [9] cannot handle this operation. Hence, SMT code for DIP is not generated. Even if the SMT code is generated by other means, the SMT has to try all possible shifts during DIP discovery making it hard for the solver.

2) Obfuscates the input operations of a multiplier: The SMT solver is not efficient in handling multiply operation since it creates potential non-linear instances. Even the SAT attack fails on the ISCAS89 circuit which models a multiplier [14]. HOST creates non-linear instances in the DIP model by obfuscating the inputs of multiplications. This creates non-linear equations over the keys. The DIP-guided SMT attack tool adds extra constraints over keys in every iteration into its initial DIP formulation to prune incorrect keys. The constraint added at every iteration is non-linear. Proving UNSAT (i.e., no DIP found) of the DIP formula becomes harder at every iteration.

Consider the C code and DFG in Figure 3. HOST obfuscates $o1$ and $o2$ with keys $k1$ and $k2$, respectively. $o1$ and $o2$ are inputs to multiplication $o3$. Assume that the SMT solver finds a DIP for the initial formulation and returns the input values as $a = 1,\ b = 2,\ c = 3,\ d = 4, m = 5,\ n = 6,\ p = 7,\ q = 8$. For these input values, the oracle gives $out = 21$. The constraint $(k1\ ?\ 3 : 11) \times (k2\ ?\ 15 : 7) = 21$ is added to the DIP in the next iteration. Keys $k1$ and $k2$ are multiplied in this constraint. Obfuscating back-to-back multipliers in a DFG creates a hard instance for SMT solvers.

C. HOST Increases # of Iterations to Thwart SMT Attack

The SAT attack works by pruning the key space by adding constraints at every iteration. The number of key entries eliminated at each iteration can be reduced by camouflaging the key input by *controlling corruptibility* [16], [18], [19]. We redefine

```
STATE_4 : begin
   if ({key[0], key[1]} == 2'b00 &&
       reg_comb_1 == rand_val_1) begin
     // Activate random control signals
     NEXT_STATE = STATE_8;
   end
   else if ({key[0], key[1]} == 2'b01 &&
       reg_comb_2 == rand_val_2) begin
     // Activate random control signals
     NEXT_STATE = STATE_10;
   end
   else begin
     // Activate correct control signals
     NEXT_STATE = STATE_9
   end
end
```

Listing 4: Spurious transitions: Snippet for FSM state 4 Fig 4

```
/*************Original code snippet**************/
STATE_2 : begin
     // control signal transfers
     NEXT_STATE = STATE_3;
end
/************Obfuscated Code snippet*************/
STATE_2 : begin
     // control signal transfers
     if(key[0] == 1'b1 &&
     reg_comb == rand_val) begin
      NEXT_STATE = STATE_x_2; // Spurious state
     end
     else begin
      NEXT_STATE = STATE_3; // Genuine state
     end
end
STATE_x_2 : begin // Spurious state S'2
    // Activate random control signals
end
```

Listing 5: Spurious state obfuscation

obfuscations *to corrupt the output for only a restricted set of inputs* in case of an incorrect key. This prunes the key space per iteration and increases the number of iterations to extract the complete key. To implement *controlled corruptibility*, we select a random combination of registers R_{τ,S_x} informed by the specifications from the datapath. Let τ be a trace[1] executed for an input to the RTL code and S_x be a state within the trace where the key verification takes place. R_{τ,S_x} can be any combination of registers updated anywhere in the trace τ prior to state S_x. R_{τ,S_x} is expected to have an indirect or direct input dependency at state S_x, where we execute the key

[1]trace is an execution path of the FSM.

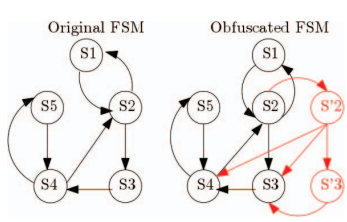

Fig. 5: FSM obfuscation uses spurious states

verification condition. In conjunction with the key verification condition, we add the condition ($R_{\tau,S_x} == rand\_val$), where $rand\_val$ is a random bitmap vector of same size as R_{τ,S_x}. If this is satisfied along with a wrong input key, we corrupt the design output using techniques described below. Since we trigger the corruption for a specific bitmap of R_{τ,S_x}, the SMT solver has to find the input which produces that bitmap for R_{τ,S_x} to eliminate the key.

1) Controlled corruptibility using spurious transitions: Spurious transitions in the FSM disrupts the functional output for an incorrect input and successful corruptibility condition. Listing 4 is the code snippet corresponding to state 4 of transition obfuscated FSM in Fig 4, where red transitions are spurious. The $4 \rightarrow 9$ transition is obfuscated with two keys *key[0,1]*. For incorrect keys and at least one successful corruptibility check among conditions (*reg_comb_1 == rand_val_1*) or (*reg_comb_2 == rand_val_2*), the FSM transitions to incorrect state (8 or 10), distorting the output. *reg_comb_1* and *reg_comb_2* represents two R_{τ,S_x} register combinations explained earlier. *rand_val_1* and *rand_val_2* are randomly chosen bitmaps for *reg_comb_1* and *reg_comb_2*.

2) Controlled corruptibility using spurious states: If the number of states in the FSM are insufficient to provide obfuscation strength, HOST embeds *spurious* states by adding random control signals that perform undesirable RTL operations. The extra states increase the space for spurious transitions and increase the number of key bits in the obfuscated design. Since the adversary is unaware of the number of FSM states in the original RTL, they cannot spot the spurious states provided they do not create livelocks or deadlocks. An FSM with red color spurious states is shown in Fig 5. Listing 5 elaborates the code segments corresponding to the original and obfuscated FSM state 2 in Fig 5, where *key[0]* = '0' triggers the correct transition. If the input is in the small subset of inputs satisfying the corruptibility condition (*(reg_comb == rand_val)*), the FSM sequence is directed to spurious state *STATE_x_2* where random control signals activate random incorrect traces in the datapath. *reg_comb* represents the R_{τ,S_x} register combination and *rand_val* is the random bitmap.

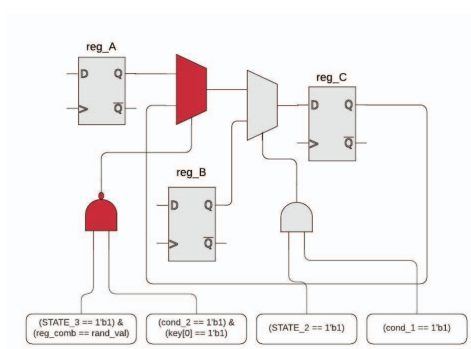

Fig. 6: HOST adds red units for datapath obfuscation

3) Controlled corruptibility using spurious register transfers: Spurious register transfers extend control signal obfus-

cation to implement controlled corruptibility. Instead of modifying a register transfer, HOST modifies the HLS-generated datapath to create register transfers on an incorrect key and successful corruptibility check. This increases the number of obfuscation key bits. Listing 6 shows the register transfers in states 2 and 3 of a sample FSM. The obfuscated snippet does a spurious register transfer $reg\_C \leftarrow reg\_A$ in case of a wrong key (key[0] = 1) and a successful corruptibility check (*reg_comb == rand_val*). The red components in the HOST-modified datapath in Fig 6 add spurious register transfers.

```
/************Original code snippet*************/
always @(posedge clk) begin
  if((cond_1 == 1'b1) & (STATE_2 == 1'b1)) begin
    reg_C <= reg_B;
  end
end
always @(posedge clk) begin
  if((cond_2 == 1'b1) & (STATE_3 == 1'b1)) begin
   //significant register transfer operations
  end
end
/************Obfuscated Code snippet************/
always @(posedge clk) begin
  if((cond_1 == 1'b1) & (STATE_2 == 1'b1)) begin
    reg_C <= reg_B;
  end
  else if ((cond_2 == 1'b1) &
  (reg_comb == rand_val)) &
  (key[0] == 1'b1)) begin
   reg_C <= reg_A; // Spurious register transfer
  end
end
always @(posedge clk) begin
  if((cond_2 == 1'b1) &
  (STATE_3 == 1'b1) & (key[0] == 1'b0)) begin
   //Correct register transfer operation
  end
end
```

Listing 6: HOST datapath obfuscation

V. Experimental Evaluation

A. Experimental Setup

HOST is implemented using Python and is integrated with the PyVerilog toolkit [15] to generate the AST representation from the Verilog RTL generated by Vivado HLS tool. The obfuscations are implemented on the AST and the obfuscated Verilog RTL is generated. The obfuscated RTL is fed to the SMT attack tool where it is first parsed using [15]. Subsequently, the rewriting method [3] generates the RTL-FSMD from the AST. We demonstrate efficacy in thwarting state-of-art SMT attack [9]. The obfuscations are evaluated on CHStone benchmarks [7]. We quantify hardware and performance overhead of the obfuscation logic. The HLS RTL used in the experiments were synthesized using Vivado HLS 2019.2 [1] targeting Kintex 7 series FPGA [17] clocked at 100 MHz. We ran the experiments on a 32 Core Intel(R) Xeon(R) CPU E5-2620 v4 (2.10 GHz) with 64GB DRAM.

B. HOST vs. TAO

In Table I, we compare the maximum number of key bits generated by TAO and HOST on benchmarks from [10]. We report constants (# Consts), conditional statements (# Brnch) and operations (# Ops) in C input and Verilog RTL generated by Vivado HLS. The maximum key bits generated by HOST and TAO are reported in the last two columns. Since the generated RTL is 10× the corresponding C code, HOST offers more scope for obfuscation. For constants, we use keys with bit-width same as target constant. A one bit key is used for each control branch and operation. TAO uses 32-bit keys for constants. For an 8-bit constant, 24 key bits are then set to zero. This happens for Viterbi (with 117 constants) and TAO reports a high number of keys. Except for Viterbi, HOST has better obfuscation opportunities at RTL than during HLS.

Benchmark	# Const		# Brnch		# Ops		# Key bits	
	TAO	HOST	TAO	HOST	TAO	HOST	TAO	HOST
GSM	4	1118	4	332	88	290	484	4678
ADPCM	5	1486	5	524	100	245	565	6574
SOBEL	2	128	11	36	2	27	110	713
BACKDROP	12	450	123	128	11	76	887	1654
VITERBI	117	150	9	47	98	31	4145	515

TABLE I: Comparative analysis: TAO vs. HOST

C. Area and Timing Impact of HOST

We consider the large benchmarks from the CHStone suite [7] to study the overhead of SMT-resilient obfuscation. Table II presents the resource overhead when adding 20% spurious states (*Spur-State*), 20% spurious transitions (*Spur-Tran*), 20% or maximum possible instances (whichever is larger) of the obfuscated multiplication and bit-shift instances (*Hard-Inst*), 20% obfuscated control signals (*Ctrl-sig*) and 20% spurious register transfers (*Spur-Reg*).

Benchmark	Spurious state(%)	Spurious trans(%)	Hard Instance(%)	Control signals(%)	Spurious Regs(%)
MIPS	3.3	0.4	6.6	4.4	5.6
AES-ENCRYPT	4.1	1	11.5	7.2	5.4
AES-DECRYPT	7.0	0.7	8.6	7.6	8.8
DFSUB	8.9	0.8	2.1	7.8	2.9
DFMUL	2.0	0.07	1.3	14.7	10.9
Average	**5.06**	**0.59**	**6.02**	**8.34**	**6.72**

TABLE II: Area overhead of SMT-resilient obfuscations.

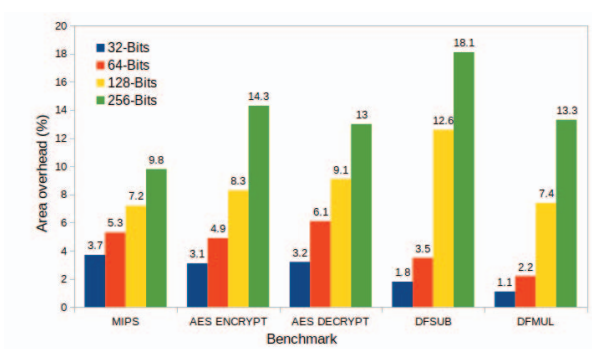

Fig. 7: Area overhead per fixed Key-size

None of the HOST obfuscations use extra instances of BRAM or DSP slices. Hence the overhead is computed using the relative number of Configurable Logic Blocks (CLB) tiles in the obfuscated design with respect to the baseline. HOST obfuscations reported <10% resource overhead and spurious transitions consume <1% extra resources. Spurious state and transition obfuscated designs reported a negligible change in Worst Negative Slack (WNS) (i.e. the critical path remained unaltered by the obfuscation logic). Spurious RTL obfuscation had a worst case change in WNS of ∼10%. Hard-instance and control signal obfuscation impacted WNS in a few cases when the obfuscation logic is in the critical path. Critical-path-aware obfuscation can obviate this. We evaluate the overhead of obfuscation logic as a function of the key size. Fig 7 shows the overhead when using SMT-resilient+non-resilient obfuscation approaches for a target key size. On an average, 32, 64, 128 and 256 bit obfuscations have an overhead of 2.6%, 4.4%, 8.9% and 13.7% respectively. Hardware overhead increases linearly with key size. Timing overhead has a similar trend.

Benchmark	Phase 1			Phase 2			Phase 3		
	#key-bits	#Iterations	Time(s)	#key-bits	#Iterations	Time(s)	#key-bits	#Iterations	Time(s)
WAKA	10	5	26	10	10	966	20	-	TO
	20	4	15	20	-	TO	-	-	-
MOTION	10	2	41	10	6	3585	20	-	TO
	15	2	1143	15	-	TO	-	-	-
ARF	10	2	473	10	-	TO	-	-	-

TABLE III: Evaluation of SMT Attack Resiliency

D. HOST Resiliency against SMT attack

The resilience of HOST obfuscated RTLs is verified using the SMT attack [9]. The SMT attack tool does not support arrays, function calls and loops with dynamic bounds (a limitation of SMT solver Z3). Benchmarks with these constructs cannot be handled by this version of SMT attack. For fair comparison, we use benchmarks from [9] and Table III reports results of HOST obfuscations in subsections IV-B and IV-C in three phases[2]. We use 10, 15, and 20 bits key for spurious transitions, states and register transfers without *controlled corruptibility* conditions. Next, we add *controlled corruptibility* from subsection IV-C while maintaining the number of key bits. Finally, we add 10 bits for the hard-instance obfuscation in IV-B. We set 10 hrs as the Time-Out (TO) for the attack. The number of iterations doubles and the run-time increases up to 87× when controlled corruptibility is considered. The attack times out in 3-out-of-5 cases when controlled corruptibility is considered and times out in the remaining 2-out-of-5 cases when hard instances+controlled corruptibility are considered.

[2]Since control signal obfuscation of subsection IV-A prevents RTL-FSMD extraction, it is not used.

VI. Conclusions

HOST is a post-synthesis RTL obfuscation technique that exploits the FSMD structure of HLS generated RTL. HOST offers better scope for high-level obfuscation in terms of key size compared to HLS-based obfuscations in TAO [10]. The hardware overhead is minimal. HOST successfully thwarts state-of-art SMT attack [9]. Since HOST operates on a scheduled RTL, a few obfuscation techniques impact the critical path and consequently reduce the operable frequency of the design. In response, a critical-path-aware post synthesis RTL obfuscation is a direction to explore.

VII. Acknowledgments

C. Karfa was partially supported by the DST, Government of India under Project CRG/2019/001300. R. Karri is supported in part by NSF 1526405, ONR grant N00014-18-1-2058, NYU CCS, and NYU CCS-AD.

References

[1] *Vivado High-Level Synthesis:"http://xilinx.com/support/download.html"*.
[2] H. Badier, J. L. Lann, P. Coussy, and G. Gogniat. Transient key-based obfuscation for HLS in an untrusted cloud environment. In *DATE*, pages 1118–1123, 2019.
[3] C. Karfa, D. Sarkar and C. Mandal. Verification of Datapath and Controller Generation Phase in High-Level Synthesis of Digital Circuits. *IEEE TCAD*, 29(f3):479–492, Mar 2010.
[4] C. Karfa, D. Sarkar, C. Mandal and C. Reade. Hand-in-hand verification of high-level synthesis. In *GLSVLSI*, pages 429–434, 2007.
[5] R. S. Chakraborty and S. Bhunia. Harpoon: An obfuscation-based soc design methodology for hardware protection. *IEEE Transactions on CAD of ICS*, 28(10):1493–1502, 2009.
[6] R. S. Chakraborty and S. Bhunia. Rtl hardware ip protection using key-based control and data flow obfuscation. In *VLSID*, page 405–410, 2010.
[7] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.
[8] S. A. Islam and S. Katkoori. High-level synthesis of key based obfuscated rtl datapaths. In *19th ISQED*, pages 407–412, 2018.
[9] C. Karfa, R. Chouksey, C. Pilato, S. Garg, and R. Karri. Is register transfer level locking secure? In *DATE*, pages 550–555, 2020.
[10] C. Pilato, F. Regazzoni, R. Karri, and S. Garg. TAO: Techniques for algorithm-level obfuscation during high-level synthesis. In *IEEE/ACM Design Automation Conference*, pages 1–6, June 2018.
[11] J. Rajendran, A. Ali, O. Sinanoglu, and R. Karri. Belling the cad: Toward security-centric electronic system design. *IEEE TCAD*, 34(11):1756–1769, 2015.
[12] J. Rajendran, H. Zhang, C. Zhang, GS. Rose, Y. Pino, O. Sinanoglu, and R. Karri. Fault analysis-based logic encryption. *IEEE Transactions on computers*, pages 410–424, 2013.
[13] J. A. Roy, F. Koushanfar, and I. L. Markov. Ending piracy of integrated circuits. *Computer*, 43(10):30–38, 2010.
[14] P. Subramanyan, S. Ray, and S. Malik. Evaluating the security of logic encryption algorithms. In *HOST'15*, pages 137–143, May 2015.
[15] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040, pages 451–460, Apr 2015.
[16] Y. Xie and A. Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE TCAD*, 38(2):199–207, 2019.
[17] Xilinx. *Xilinx Kintex 7 series FPGA configuration manual*, 2018.
[18] M. Yasin, B. Mazumdar, J. Rajendran, and O. Sinanoglu. Sarlock: Sat attack resistant logic locking. In *HOST*, pages 236–241, 2016.
[19] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, JV Rajendran, and O. Sinanoglu. Provably-secure logic locking: From theory to practice. In *ACM SIGSAC CCS*, 2017.
[20] M. Yasin, C. Zhao, and J. J. Rajendran. Sfll-hls: Stripped-functionality logic locking meets high-level synthesis. In *ICCAD*, pages 1–4, 2019.