DTjRTL: A Configurable Framework for Automated Hardware Trojan Insertion at RTL

Ruochen Dai
ruochendai@ufl.edu
University of Florida
Gainesville, FL, USA

Zhaoxiang Liu
zxliu@ksu.edu
Kansas State University
Manhattan, KS, USA

Orlando Arias
orlando_arias@uml.edu
University of Massachusetts Lowell
Lowell, MA, USA

Xiaolong Guo
guoxiaolong@ksu.edu
Kansas State University
Manhattan, KS, USA

Tuba Yavuz
tuba@ece.ufl.edu
University of Florida
Gainesville, FL, USA

ABSTRACT

Shifts in the IC supply chain have necessitated outsourcing design or fabrication to third-party vendors, introducing various hardware security issues, notably Hardware Trojans (HTs) as a prominent risk. The research in detecting and preventing HTs faces challenges due to the lack of standardized benchmarks and measurements. This paper introduces a framework to automatically generate dynamic functional HTs in a configurable and systematical manner at Register Transfer Level (RTL). The objective is not to produce HTs that are difficult to activate but to systematically create a diverse set of HT designs. This approach serves dual purposes: it aids the research community in testing their detection frameworks and facilitates buggy design benchmark creation for competitive exercises between blue and red teams. Our framework accepts RTL designs and configuration parameters, automating the generation of HT-inserted designs at RTL. We present an evaluation of the generated HT designs focusing on hardware cost overhead and post-synthesis survivability by verifying HT presence at both RT and gate levels. Results indicate that HTs employing only combinational logic are easier to optimize away but result in lower overhead compared to HTs that incorporate additional sequential logic.

CCS CONCEPTS

• **Security and privacy → Malicious design modifications**.

KEYWORDS

Hardware security, Hardware Trojan, Automated Benchmarking, RT-Level Security

ACM Reference Format:
Ruochen Dai, Zhaoxiang Liu, Orlando Arias, Xiaolong Guo, and Tuba Yavuz. 2024. DTjRTL: A Configurable Framework for Automated Hardware Trojan Insertion at RTL. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

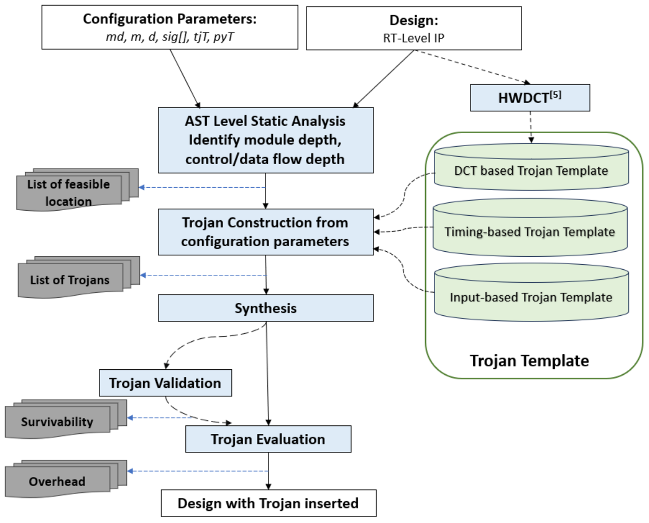

Configuration Parameters:
md, m, d, sig[], tjT, pyT
Design:
RT-Level IP
HWDCT[5]
AST Level Static Analysis Identify module depth, control/data flow depth
List of feasible location
Trojan Construction from configuration parameters
List of Trojans
Synthesis
DCT based Trojan Template
Timing-based Trojan Template
Input-based Trojan Template
Trojan Template
Trojan Validation
Survivability
Trojan Evaluation
Overhead
Design with Trojan inserted

Figure 1: Overall workflow of DTjRTL.

1 INTRODUCTION

The supply chain for integrated circuits (IC) is experiencing a pivotal shift from a traditionally vertically integrated model to a horizontally oriented one which involves multiple collaborative companies at different stages of the IC production process. Although this shift provides economic and logistical benefits, it also opens up numerous security vulnerabilities, with hardware Trojans standing out as a significant concern. Recent years have seen numerous research efforts [2, 5, 16] aimed at detecting hardware Trojans, yet the challenge remains in selecting appropriate benchmarks for validating these detection methodologies.

Standardized benchmarks serve as a crucial baseline for evaluating the effectiveness of HT detection methods. A common strategy [16, 19] includes utilizing static benchmarks from the TrustHub platform [17], which comprises 106 HT-infected benchmark circuits. Notably, these 106 hardware Trojans are derived from only 7 distinct designs. This contribution aids in standardizing HT detection tests but faces limitations due to its static nature, where Trojan locations and triggers are predetermined. This specificity may lead to detection methods being overly tailored to these benchmarks rather than being applicable to a broader range of hardware

Table 1: Objective of each Configuration Parameter.

Param.	Objective
md	Module depth, determine candidate modules to insert the HT
m	AST analysis method, either structural control-flow or signal-dependent control-/data- flow analysis
d	control/data flow distance, determine the location of HT
sig[]	trigger signal set, determine the set of signal used to construct trigger
tjT	trigger type, either combinational, sequential, or template HT (DCT, Tim, Inp based)
pyT	payload type, determine HT functionality, either AND, OR, or XOR operation

Trojans. Additionally, the suite's inability to quickly adapt to new Trojan types poses a further limitation. In instances where existing benchmarks do not cover a particular Trojan type, researchers are compelled to create specific Trojans [5], raising concerns about the benchmarks' quality. Authors in [11] further show that only 3 out of 83 benchmark designs from TrustHub can be considered as actual hardware Trojans while the others are proven to be ineffective, which shows the challenges in developing realistic HTs and the importance of automated and parametric HT generation that can allow users apply various criteria to the candidate HTs.

Recently, numerous studies [1, 3, 4, 6, 10, 10, 15, 21] have aimed at automating the generation of benchmark suites, with a significant focus on inserting HTs at the gate-level [1, 3, 4, 10, 15, 21]. Those approaches introduce two main issues: Firstly, gate-level HT insertion, relying on structural characteristics or Sandia Controllability/Observability Analysis Program values to place HTs at rare-activated nodes, uses controllability and observability scores to gauge the difficulty of manipulating each node. However, this method's reliance on gate-level semantics complicates its application to RTL HT insertion, especially for diverse HT templates, thus undermining efforts to systematically introduce HTs. Furthermore, the rise of AI-based HT detection frameworks [19] underscores the growing demand for a substantial collection of HT-embedded designs for training, accentuating the necessity for HT-inserted designs at the RTL.

To establish a more standardized and systematic benchmark suite at RTL, this paper introduces a framework, DTjRTL, for the automated and configurable insertion of dynamic functional HTs at RTL. This framework enables users to insert various HTs into their hardware designs, allowing the selection of either combinational or sequential HT triggers, as well as the choice of payload location and function (AND, OR, XOR). DTjRTL also permits the insertion of template-based HTs, currently supporting three types: Input, Timing, and Don't Care Transition-based templates. Figure 1 outlines the comprehensive process of DTjRTL, which processes RTL hardware designs alongside configuration parameters to produce designs inserted with HTs. The framework leverages static analysis to identify module depth and control/data flow depth, informing the payload location selection. Subsequently, it generates potential HT instances based on user-defined parameters, synthesizes them to gate-level representations, optionally validates HT synthesis survivability, and evaluates the hardware cost overhead. The main contributions of this paper are:

- The first effort to automate and generate dynamic functional HTs in a configurable and systematic way at RTL.
- Implementation of proposed approach in an automated tool, DTjRTL[1], which enables the scalability of HT insertion.
- Evaluation of the automatically generated HT designsin terms of hardware cost overhead and synthesis survivability.

2 BACKGROUND AND RELATED WORK

2.1 Threat Model

DTjRTL focuses on the insertion of Trojans in the RTL for studying prevention and mitigation techniques. As such, we follow the model where a Trojan may be inserted into the hardware design by a third-party intellectual property vendor. We also assume that the synthesis tool is trustworthy.

2.2 Trojan Insertion

Hardware Trojan can be inserted at RT- [1, 6, 10, 22], gate- [1, 3, 4, 10, 15, 21], or transistor- level [10]. The pre-synthesis HT insertion in [1] is applicable to combinational logic only. For the HT insertion at RT-Level, [22] utilizes unused circuit identification (UCI) [8] techniques to generate hard to trigger HTs that can evade existing detection techniques. They rely on a specific coding style and trigger input selection. Additionally, signal controllability is examined from the attacker's perspective . [6] propose a set of RTL HT benchmarks injected in a RISC-based pipelined microprocessor core, lacking configurability and automation. Cruz et.al. [4] firstly proposed a tool flow for inserting custom dynamic HTs with validated payload an trigger conditions in gate-level designs by identifying rare internal nodes through functional simulation. However, functional simulation can provide only an estimation of the switching activity of the internal nodes, and its accuracy is closely related to the number and quality of test patterns applied to the design inputs. Thus [21] improves [4] by using a highly configurable generation platform based on transition probability modeled by geometric distribution to identify the rarely activated nodes.

Authors in [3] propose an automated HT insertion framework using a greedy approach and SCOAP [7] reduction method that can survive from both circuit structural feature-based and SCOAP-based detection methods at gate-level. Taint [10] targets FPGA designs and automatically inserts HT at either RTL, gate-level, or post-map netlist by using a multiplexer to connect original and HT payloads with the activation circuit, which is defined by author and not configurable. [15] utilizes Reinforcement Learning as a means to identify rare nodes and automate the HT insertion process to eliminate the inherent human biases at gate-level netlist.

3 APPROACH

3.1 Overall workflow

Figure 1 illustrates the comprehensive process of DTjRTL, accepting a RTL hardware design in Verilog or SystemVerilog alongside a configuration parameter set, to automatically generate a design with HT integration with their synthesis survivability and hardware cost overhead. Algorithm 1 delineates the four pivotal steps. Initially, line 2 obtains the *DepthSet*, encompassing module depth, structural and

[1]https://github.com/RuochenDai78/DTjRTL

signal-dependent control/data-flow depth, and (LHS, RHS) signal pairs across various depths, through HDL parsing into an Abstract Syntax Tree (AST) utilizing Verilator [18]. Subsequently, line 3 employs the configuration parameters C and $DepthSet$ to formulate the HT inserted design T_rtl and an assertion map $AssertionMap$ at the RTL, with each parameter's elaborate clarification presented in Table 1, where the first three parameters (md, m, d) determine the location of the Trojan, and the remaining three parameters $(sig[], tjT, pyT)$ set the structure of the Trojan. Line 4 proceeds to synthesize the original design P into gate-level representation, which will be used for synthesis survivability analysis (lines 7-11) and for calculating the hardware cost overhead (OH) (line 21). Our framework generates assertions based on the payload logic to support the synthesis survivability, which is an optional aspect of our framework. If it is chosen, $SS == True$, the validation is performed based on whether a test bench TB is provided or not. If it is provided, functional simulation is used. Otherwise, we use fuzzing, e.g., [16], for validation. Both the RTL and gate-level versions of the HT are validated. If both $valid\_rtl$ and $valid\_gate$ are affirmed, indicating HT's synthesis survival, the HT is deemed valid (line 16). If synthesis survivability is not chosen, each candidate HT is deemed valid. Subsequently, OH is calculated based on gate count, and HTs with OH below threshold τ are included in the return set (line 21). Finally, HTs that are deemed valid and with low overhead w.r.t. τ are returned.

3.2 Structural Control-Flow Analysis

Structural Control-Flow Analysis (SCFA) delves into the structural features of a system, highlighting loops, branches, and execution paths. It assesses the system to generate a detailed enumeration of assignments, each paired with a distinct Structural Control-Flow depth (SCFd). This SCFd value quantifies the depth of each assignment within the system's control-flow hierarchy, providing a nuanced understanding of its structural context.

Listing 1 provides an illustration, wherein SCFA identifies all assignments within various code segments. The assignment to A_is_Max at line 6 represents a continuous assignment outside any branches, assigning its SCFd as 0. Conversely, the assignment to $Result$ at line 10 is situated within an `if` branch (Line 9), hence its SCFd is 1. Regarding the assignments to $Result$ at lines 12 and 13, as they are inside a nested branch, their SCFds are evaluated as 2.

Listing 1: A simple design in Verilog to demonstrate Structural control-flow analysis.

```
1   module scfa_demo(A, B, Result);
2   input [1:0] A, B;
3   output reg Result;
4
5   wire A_is_Max;
6   assign A_is_Max = &(A & 2'b11);
7
8   always @(A or B) begin
9     if (A > B) begin
10      Result = 1;
11    end else begin
12      if (A_is_Max) Result = 1;
13      else Result = 0;
14    end
15  end
16  endmodule
```

Algorithm 1: RTL Dynamic Trojan Insertion.

```
Input: $P$: HW Design, $C$: Configuration parameter, $\tau$:
       Threshold, $SS$: Boolean, $TB$ : Test Bench
Output: Set of ($T$: Trojan inserted HW design, $OH$: Trojan
        design overhead)
1  $T \leftarrow \emptyset$;
2  $DepthSet \leftarrow ASTAnalyzer(P, C)$;
3  $(T\_rtl, AssertionMap) \leftarrow$
     $TjConstructGenAssertion(C, DepthSet)$;
4  $orig\_gate \leftarrow Synthesize(P)$;
5  for each $T\_rtl_i \in T\_rtl$ do
6      $T\_gate_i \leftarrow Synthesize(T\_rtl_i)$;
7      if $SS = true$ then
8          $A \leftarrow AssertionMap[T\_rtl_i]$;
9          $(T'\_rtl_i,\ T'\_gate_i) \leftarrow$
             $AddAssertion(T\_rtl_i, T\_gate_i, A)$;
10         if $TB \neq \perp$ then
11             $(valid\_rtl_i,\ valid\_gate_i) \leftarrow$
                 $FuncSim(T'\_rtl_i,\ T'\_gate_i)$;
12         end
13         else
14             $(valid\_rtl_i,\ valid\_gate_i) \leftarrow$
                 $Fuzz(T'\_rtl_i,\ T'\_gate_i)$;
15         end
16         $valid_i \leftarrow (valid\_rtl_i\ and\ valid\_gate_i)$;
17     end
18     else
19         $valid_i \leftarrow true$
20     end
21     $overhead_i \leftarrow OverheadCalc(orig\_gate,\ T\_gate_i)$;
22     $T \leftarrow T\ \cup \{(T\_rtl_i, overhead_i, valid_i)\}$;
23 end
24 return $\{(t, o) \mid (t, o, v) \in T\ \wedge\ o < \tau\ \wedge\ v = true\}$
```

3.3 Signal-dependent Data/Control-flow Analysis

Signal-dependent Data/Control-flow Analysis (SD_D/CFA) examines signal dependencies across modules. We evaluate how signals are driven by the input, i.e. the correlation between the input stimulus with the other signals. The correlation can be calculated through the information flow tracking (IFT) technique. To perform IFT, we first build the directed graph $G = (V, E_c \cup E_d)$ on AST with [12]. Here, V denotes the set of hardware-declared signals, E_c captures the control logic connection such as *If* and *Case Statement*, while E_d represents various assignment types including *blocking* and *non-blocking assignments*. SD_DFA calculates the paths between signals on graph (V, E_d) and SD_CFA considers the paths on graph (V, E_c).

Taking Listing 2 as an example, we conduct SD_CFA on signal *top_ret*, whose shortest control depth is 2 including conditions $A > B$ and *inter*. The SD_DFA on signal A_is_Max has data flow depth 2 with path $top\_A \rightarrow a\_inter \rightarrow A \rightarrow A\_is\_Max$.

Table 2: Comparison of number of potential Trojan insertion locations between structural and signal-dependent AST analysis. `st` = structural, `sd` = signal-dependent, `cfa` = control-flow analysis, `dfa` = data-flow analysis.

Design	m	# of Troj locations at d											Total Insertion Time (s)	Source
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	***total***		
AES	st_cfa	179	195	24	7	-	-	-	-	-	-	405	1.47	OpenCores [14]
	sd_cfa	3	97	143	5	-	-	-	-	-	-	248	1.35	
	sd_dfa	-	81	94	30	11	12	8	6	5	1	248	1.43	
RS232	st_cfa	15	29	26	5	-	-	-	-	-	-	75	0.51	TrustHub [17]
	sd_cfa	-	35	55	4	-	-	-	-	-	-	94	0.54	
	sd_dfa	-	37	53	4	-	-	-	-	-	-	94	0.54	

Listing 2: Example to demonstrate Signal-dependent Data/Control-flow Analysis.

```
module top(top_A, top_B, top_ret);
input [1:0] top_A,top_B;
output [1:0] wire top_ret;
wire inter;
wire [1:0] a_inter;
scfa_demo scfa_demo_i(
.A(a_inter),
.B(top_B),
.Result(inter)
);
assign top_ret = inter ? 2'b11:2'b00 ;
assign a_inter=top_A + 2'b1;
endmodule
```

3.4 Trigger type of Trojan

As indicated by the configuration parameter tjT in Table 1, DTjRTL facilitates the insertion of both combinational, sequential, and template HTs.

Combinational Trojans. These are triggered by a distinct combination of signals meeting a specific criterion, directly affecting the embedded logic gates and interconnects. Their activation is contingent on the simultaneous presence of certain input conditions, independent of previous signal sequences.

Sequential Trojans. These rely on a series of events or states encountered by the design overtime for activation. This dependency on sequential conditions and state history renders them stealthier and more difficult to detect, as their activation may mimic regular operational patterns, remaining undetected until executing their intended malicious functions.

Trojan Template. To address the challenge of integrating new Trojan configurations, DTjRTL introduces a mechanism that enables users to incorporate custom Trojan structures. In this study, we demonstrate this capability by integrating three distinct types of Trojan templates: Don't Care Transition (DCT)-based, Timing-based, and Input-based Trojans.

1) DCT-based Trojan Template. This template leverages the unused or don't care conditions within a Finite State Machine in the design as triggers for Trojan activation. The insertion of such HT involves the use of HWDCT [5] for preprocessing the design to identify these DCTs.

2) Timing-based Trojan Template. This approach enables the embedding of HT that are activated by distinct timing events or scenarios, including the execution sequence timing of operations. An example of such utilization involves activating the HT after a predetermined number of encryption processes within AES core.

3) Input-based Trojan Template. This template is designed to insert HT that become active when a particular user-specified input or internal signal attains a predetermined significant value. For instance, within a 128-bit AES encryption core, the Trojan is activated if $key == 128'h0123\_4567\_89ab\_cdef\_0000\_0000\_0000\_0000$.

4 EVALUATION

To illustrate the capabilities of DTjRTL, AES [14] and RS232 [17] cores were tested on a system powered by an Intel CPU E5-2698 v3 @ 2.30GHz. DTjRTL accepts RTL HDL coded in Verilog/SystemVerilog, outputting Trojan-inserted RTL code along with hardware cost overhead and synthesis survivability. We use open-source synthesis tool YOSYS [20] to synthesize RTL design, and collect total number of gates as the metric for HT Evaluation. HT validation is conducted via functional simulation using a testbench template that instantiates both the original and HT-embedded designs, incorporating assertions from *AssertionMap* as described in Algorithm 1. This process is compatible with both the open-source Icarus Verilog [9] and the commercial Modelsim [13] simulators. Additionally, users may use coverage-guided simulation as outlined in HW-Fuzz [16].

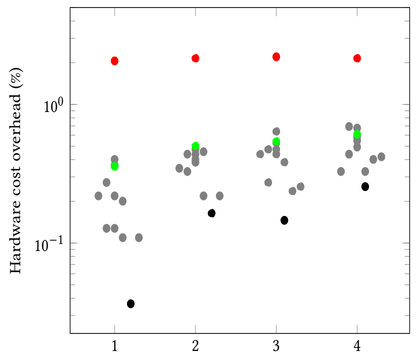
4.1 Effect of AST analysis method `m` and depth `d`

The AST analysis can be categorized into structural (`st`) or signal-dependent (`sd`) approaches, focusing solely on identifying potential Trojan locations rather than defining the Trojan's structure. Hence, our comparison is limited to the quantity of feasible Trojan insertion points in terms of method `m` and control (`cfa`) and data (`dfa`) flow analysis depth `d`.

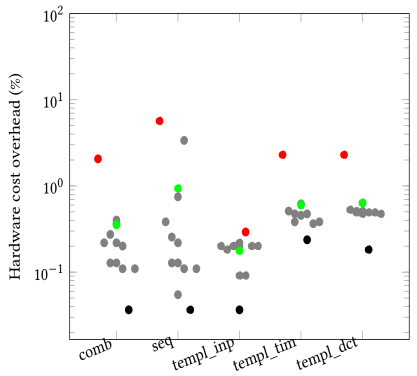
Table 2 outlines the disparities in the potential Trojan insertion locations identified by both methodologies across varying depths (`d`), showcasing a pronounced variance. The aggregate count of Trojan locations differs between `st` and `sd` for distinct benchmarks, a variation attributable to their unique processing approaches: `st` initiates with the identification of *md*, followed by checking each module within *md* for branch information, whereas `sd` assesses depth relative to input ports, taking into account the design's hierarchy. For `cfa` and `dfa`, although the total Trojan location count remains uniform, the number of identified locations at individual depths differs, highlighting the distinction between `cfa` and `dfa`.

4.2 Hardware cost overhead evaluation

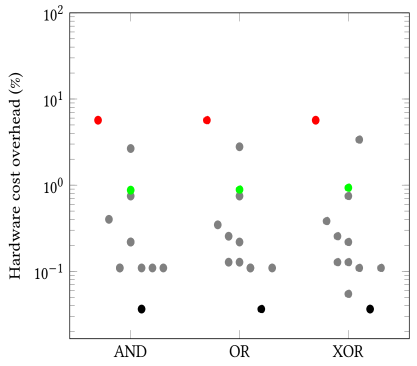
In this section, we focus on the AES core, specifically the *key_schedule* module, to illustrate the hardware cost overhead. This choice is motivated by the presence of don't-care transitions within the *key_schedule* module of AES, identifiable both before and after synthesis, as documented by HWDCT. To specifically target the



(a) Trigger signal set (`sig[]`) size v.s. overhead



(b) Trigger type (`tjT`) size v.s. overhead



(c) Payload type (`pyT`) size v.s. overhead

Figure 2: Comparing the hardware cost overhead with `sig[]`, `tjT`, and `pyT` for AES design, Figure 2a, 2b, and 2c use the configuration parameter $C = \{md = 1,\ m = scf,\ d = 0,\ tjT = comb,\ pyT = XOR\}$, $C = \{md = 1,\ m = scf,\ d = 0,\ pyT = xor\}$, and $C = \{md = 1,\ m = scf,\ d = 0, sig[] = sbox\_data\_i,\ tjT = seq\}$ respectively. Gray dots denote the cell increase rate for each Trojan instance, black, red, and green dots indicate the group's minimum, maximum, and average rate, respectively.

key_schedule module, the configuration parameter C is set to $md = 1,\ m = st\_cfa,\ d = 0$. Under this configuration, DTjRTL is able to generate **12** potential Trojan locations for *key_schedule*.

4.2.1 Effect of trigger signal set (`sig[]`*) size.* Figure 2a shows the hardware cost overhead of a list of Trojans-inserted design generated with configuration parameter md = 1, m = structural control-flow analysis, d = 0, tjT = combinational, pyT = OR, and the size of trigger signal set $sig[]$ ranges from 1 to 4. The minimum average overhead rate exhibits an increase with the increasing `sig[]` size, attributed to the escalating number of gates required to connect trigger signals as `sig[]` size expands. Conversely, the maximum overhead rate does not undergo a substantial rise, since the additional gates introduced by the Trojan are relatively minor compared to the overall gate count in the hardware design. Therefore, it is concluded that **the hardware cost overhead for the inserted Trojan increases as the size of `sig[]` increases**.

4.2.2 Effect of trigger type `tjT`. For Trojans employing varied trigger types, the configuration parameter C exhibits significant differences. For instance, the size of $sig[]$ must be specified for a combinational trigger, while the content of $sig[]$ is crucial for a sequential trigger. This variability complicates establishing a standard baseline for comparing the impact of trigger types. Therefore, we adopt a generalized approach to assess trigger type effects, opting for the simplest and most rational configuration parameters to create Trojan-embedded designs. Concretely, we set the size of `sig[]` to 1 for combinational Trojans, select an 8-bit signal for sequential and input-based template Trojans, and a 4-bit counter for timing-based template Trojans. According to Figure 2b, combinational and input-based template triggers incur lower overheads across maximum, minimum, and average measures, primarily because these triggers solely rely on logic gate cells, unlike other triggers that incorporate both logic gates and sequential elements like flip-flops. Additionally, template-based Trojans exhibit a more compact distribution of Trojan instances within a group compared to non-template Trojans. This compactness is attributed to the fixed nature of trigger structures in template-based implementations, where only the payload location varies, unlike non-template Trojans that may select different signals for trigger construction randomly. Consequently, it is deduced that **HTs involving sequential logic generally incur higher overheads than those based on combinational logic, and the variance in overhead for template-based HTs is less than that of non-template HTs**.

4.2.3 Effect of payload type `pyT`. As indicated in Table 1, users have the option to designate the functional payload type as either AND, OR, or XOR, with the outcomes depicted in Figure 2c. The configuration parameters $md,\ m,\ d$ are maintained as outlined in Section 4.2.1, setting $sig[]$ to sbox_data_i, an 8-bit input to the sbox, tjT to sequential, and pyT for AND, OR, and XOR. Trojan gets activated after observing a specific sequence of the sbox_data_i. The findings reveal minor variations among the Trojan instances across these three categories. Nonetheless, there is no substantial disparity in the maximum, minimum, and average hardware cost overhead among these groups. This is attributed to the overhead calculation based on the gate count, where altering the payload type changes the gate type but not the quantity of gates. Consequently, it is determined that **the hardware cost overhead induced by the inserted Trojan is minimally affected by the payload type**.

4.3 Trojan validation

In this section, Trojan validation is conducted by verifying the detectability of inserted Trojans both before and after synthesis through functional simulation, with the exception of DCT-based template Trojans, which are validated using HWDCT [5].

As shown in Table 3, all inserted Trojan designs are identifiable at the RTL. However, Trojans based on *seq*, *templ_tim*, and *templ_dct* exhibit a higher survival rate compared to those based on *comb* and *templ_inp*. This disparity is attributed to the former group incorporating sequential elements, such as flip-flops or latches, which possess internal states. These components' inherent complexity renders them less susceptible to synthesis optimizations due to their reliance on event sequences that synthesis tools might not readily predict or observe. Furthermore, synthesis tools primarily aim to enhance timing and minimize logic footprint, making sequential

Table 3: Trojan validation for different trigger types for key schedule module inside AES. tjT=trigger type, T_inst=Generated Trojan instance, T_rtl=Trojan validated at RTL, T_gate=Trojan validated at gate level.

tjT	# of T_inst	# of T_rtl	# of T_gate	survival rate
comb		12	7	0.58
seq		12	11	0.92
templ_inp	12	12	5	0.42
templ_tim		12	11	0.92
templ_dct		12	11	0.92

logic, essential for control flow and state preservation, less prone to optimization. Conversely, Trojan structures reliant on combinational logic, like those based on *comb* and *templ_inp*, face more direct optimization, allowing synthesis tools to more efficiently eliminate redundant or non-essential pathways.

4.4 Comparison with Trojan-inserted RS232 from TrustHub

Figure 3 illustrates the maximum, minimum, and average hardware cost overhead for six sets of designs, featuring one from TrustHub and five generated by DTjRTL, encompassing all possible (md, d) combinations as indicated on the X-axis. The grey numerals above the blue line indicate the number of Trojan instances per category, revealing that TrustHub provides 14 unique Trojan-embedded RS232 versions. Upon examination, these are further categorized into four groups based on the trigger mechanisms described in this work: sequential, combinational, input, and timing-based. Consequently, we generated Trojans of the aforementioned types, culminating in 300 unique Trojan instances. This figure corresponds to the 75 potential Trojan locations identified (as shown in Table 2), multiplied by the four types of triggers, yielding a total of 75*4=300 Trojan instances, aligning with our observations. The findings indicate that **Trojan designs generated by DTjRTL exhibit lower maximum, minimum, and average hardware cost overhead compared to those from TrustHub**. This variance may result from variations in payload disparity or the specific HDL code used to define the trigger mechanisms.

5 CONCLUSION

This paper introduces an automated framework for inserting dynamic functional hardware Trojans at RTL, aiming to create a more standardized and systematic RTL benchmark suite. Our evaluation indicates that HTs based solely on combinational logic tend to incur lower overhead but also have a reduced likelihood of survivability compared to HTs incorporating sequential elements. Additionally, a comparison between Trojan-embedded RS232 designs from TrustHub and those generated by DTjRTL revealed that our generated HTs exhibit lower overhead. Future work will focus on expanding DTjRTL to include multi-Trojan insertion and to evaluate the controllability and observability of the generated HTs.

ACKNOWLEDGMENTS

This project has been partially funded by NSF Awards 2019283, 2312982, and 2019310.

REFERENCES

[1] Sarah Amir et al. 2018. Development and evaluation of hardware obfuscation benchmarks. *Journal of Hardware and Systems Security* 2 (2018), 142–161.

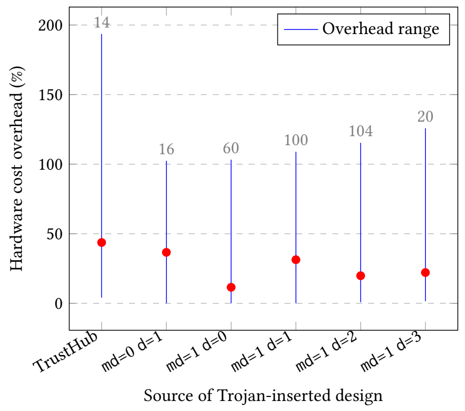


Figure 3: Comparing the Trojan-inserted RS232 design from TrustHub and DTjRTL-generated, all data are compiled with configuration parameter $C = \{m = scf,\ pyT = xor\}$. For all six groups, red dots indicate average overhead, grey number above blue line indicates # of Trojan instances in each group.

[2] Swarup Bhunia et al. 2014. Hardware Trojan attacks: Threat analysis and countermeasures. *Proc. IEEE* 102, 8 (2014), 1229–1247.
[3] Chi-Wei Chen et al. 2022. A Hardware Trojan Insertion Framework against Gate-Level Netlist Structural Feature-based and SCOAP-based Detection. In *2022 IEEE 65th International Midwest Symposium on Circuits and Systems*. IEEE, 1–5.
[4] Jonathan Cruz, Yuanwen Huang, Prabhat Mishra, and Swarup Bhunia. 2018. An automated configurable Trojan insertion framework for dynamic trust benchmarks. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1598–1603.
[5] Ruochen Dai and Tuba Yavuz. 2022. A Symbolic Approach to Detecting Hardware Trojans Triggered by Don't Care Transitions. *ACM Trans. Des. Autom. Electron. Syst.* (aug 2022). https://doi.org/10.1145/3558392
[6] Aleksa Damljanovic, Annachiara Ruospo, Ernesto Sanchez, and Giovanni Squillero. 2021. A benchmark suite of RT-level hardware trojans for pipelined microprocessor cores. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 51–56.
[7] Lawrence H Goldstein and Evelyn L Thigpen. 1980. SCOAP: Sandia controllability/observability analysis program. In *Proceedings of the 17th Design Automation Conference*. 190–196.
[8] Matthew Hicks, Murph Finnicum, Samuel T King, Milo MK Martin, and Jonathan M Smith. 2010. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 159–172.
[9] Icarus Verilog. 2002. https://github.com/steveicarus/iverilog.
[10] Vinayaka Jyothi, Prashanth Krishnamurthy, Farshad Khorrami, and Ramesh Karri. 2017. Taint: Tool for automated insertion of trojans. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 545–548.
[11] Christian Krieg. 2023. Reflections on Trusting TrustHUB. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.
[12] Zhaoxiang Liu et al. 2022. Inter-IP malicious modification detection through static information flow tracking. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 600–603.
[13] ModelSim. 2021. https://eda.sw.siemens.com/en-US/ic/modelsim/.
[14] OpenCores. [n. d.]. https://opencores.org/.
[15] Amin Sarihi, Ahmad Patooghy, Peter Jamieson, and Abdel-Hameed A Badawy. 2022. Hardware trojan insertion using reinforcement learning. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 139–142.
[16] Timothy Trippel et al. 2022. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*. 3237–3254.
[17] TrustHub. [n. d.]. https://www.trust-hub.org/#/home.
[18] Verilator. [n. d.]. https://www.veripool.org/verilator/.
[19] Rozhin Yasaei et al. 2022. Hardware trojan detection using graph neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).
[20] YOSYS. 2013. https://github.com/YosysHQ/yosys.
[21] Shichao Yu, Weiqiang Liu, and Maire O'Neill. 2019. An improved automatic hardware trojan generation platform. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 302–307.
[22] Jie Zhang and Qiang Xu. 2013. On hardware trojan design and implementation at register-transfer level. In *2013 IEEE international symposium on hardware-oriented security and trust (HOST)*. IEEE, 107–112.