NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A COMPUTER SCIENTIST'S EVALUATION OF PUBLICALLY AVAILABLE HARDWARE TROJAN BENCHMARKS

by

Scott M. Slayback

September 2015

Thesis Advisor:	Theodore Huffmire
Second Reader:	Mark Gondree

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE A COMPUTER SCIENTIST'S EVALUATION OF PUBLICALLY AVAILABLE HARDWARE TROJAN BENCHMARKS			**5. FUNDING NUMBERS**
6. AUTHOR(S) Slayback, Scott M.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Dr. Hassan Salmani and Dr. Mohammed Tehranipoor have developed a collection of publically available hardware Trojans, meant to be used as common benchmarks for the analysis of detection and mitigation techniques. In this thesis, we evaluate a selection of these Trojans from the perspective of a computer scientist with limited electrical engineering background. Note that this thesis is also intended to serve as a supplement to the existing documentation, since it provides a thorough description of each benchmark. This description presents a detailed analysis of each Trojan's activation conditions and post-activation activity. In addition, we describe the difficulties we encountered in synthesizing and simulating each Trojan, and, where possible, provide solutions to those difficulties.

14. SUBJECT TERMS building security in, design for trust, hardware intellectual property cores, Hardware Oriented Security and Trust, hardware synthesis, hardware Trojans, HDL, inherently trustworthy systems, malicious hardware, reconfigurable hardware, secure interfaces, security education, trustworthy system development, Vivado			**15. NUMBER OF PAGES** 165
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A COMPUTER SCIENTIST'S EVALUATION OF PUBLICALLY AVAILABLE HARDWARE TROJAN BENCHMARKS

Scott M. Slayback
Civilian, Scholarship for Service
B.S., Gonzaga University, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Author: Scott M. Slayback

Approved by: Theodore Huffmire
Thesis Advisor

Mark Gondree
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Dr. Hassan Salmani and Dr. Mohammed Tehranipoor have developed a collection of publically available hardware Trojans, meant to be used as common benchmarks for the analysis of detection and mitigation techniques. In this thesis, we evaluate a selection of these Trojans from the perspective of a computer scientist with limited electrical engineering background. Note that this thesis is also intended to serve as a supplement to the existing documentation, since it provides a thorough description of each benchmark. This description presents a detailed analysis of each Trojan's activation conditions and post-activation activity. In addition, we describe the difficulties we encountered in synthesizing and simulating each Trojan, and, where possible, provide solutions to those difficulties.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION AND MOTIVATION 1

II. RELATED WORK 7

III. METHODOLOGY 11
- A. GETTING THE BENCHMARKS 11
- B. SOURCE CODE ANALYSIS 12
- C. SETTING UP THE ENVIRONMENT 13
- D. SYNTHESIZING AND VIEWING A CIRCUIT 14
- E. VIEWING SCHEMATICS 16
- F. SIMULATION 17

IV. BENCHMARKS 21
- A. TROJANS IN AES_128 21
- B. TROJANS IN BASICRSA 57
- C. TROJANS INSERTED IN THE REGISTER TRANSFER LEVEL (RTL) OF RS232 66
- D. TROJANS INSERTED IN THE GATE LEVEL OF RS232 86

V. CONCLUSION 129
- A. SUMMARY 129
- B. FUTURE WORK AND LESSONS LEARNED 131
- C. ELEMENTS OF FUTURE BENCHMARKS 132

APPENDIX. RESOURCES 135

LIST OF REFERENCES 137

INITIAL DISTRIBUTION LIST 139

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Selection of a resource from the Trust Hub site. The button labeled "Download (ZIP)" will directly download an archive file containing all of the resources provided by Salmani et al. for the selected Trojan. Note that some Trojans are stored as RAR archives. The "Learn More" link will lead to a dedicated page for the selected Trojan. Note that the b19 Trojans are stored as multi-part archives, and it is necessary to visit the dedicated page and download parts from the "supporting files" section. 12

Figure 2. Layout of the Vivado Main Screen. Synthesis, Simulation and schematics are opened from the flow navigator on the left. The window on the far right shows the results of these commands. In this case, the window shows a waveform diagram resulting from simulation. Note that the results window can be popped out and viewed separately from Vivado's main window. The central windows allow some customization of the results window display. Errors and other messages are reported in the bottom window..15

Figure 3. Components commonly displayed in a Vivado Schematic. At left is an example of a custom module. This module is defined as `expand_key_128` in the Verilog file `aes_128.v`. The specific instance is named `a1`. Note that clicking the + sign in the top left corner will produce an expanded view of this module, including all internal components. The middle item is a sample register, used to store values that are relevant to the circuit across multiple clock cycles. The stacking effect is a visual cue provided to represent a bundled multi-bit register. It is possible to unbundle this collection and produce a schematic with a separate register for each bit. The rightmost figure demonstrates Vivado's representation of primitive gates. Note that this XOR gate also handles multiple bits, but that no visual cue is given...16

Figure 4. Part of the waveform diagram generated by Vivado from `test_aes_128.v` and the Trojan-free version of the AES circuit. Note that Vivado does not support changing the font size or the background color of the waveform area. Both features have been requested on the Xilinx support forums. Also note that the 1-bit input `clk` is represented by graphical lows and highs, but multi-bit inputs are represented by numerical values. For clarity, we have edited the colors of other waveforms in this document using external imaging software. This one has been left to demonstrate Vivado's default settings..................................18

Figure 5. This schematic--generated from `AES-T700`--represents the structure of a typical AES Trojan from the collection. Note the modules for the trigger (`Trojan_Trigger`) and the Trojan functionality (`TSC`). These inclusions share common inputs with the Trojan-free `aes_128` module, but they do not alter that module's internal operation. 22

Figure 6. The input region of the Trojan-free AES implementation, complete with a world view of the circuit for context. The primary inputs to this circuit are state and key. Note that state and key are immediately XORed before the first round of the encryption process begins. 23
Figure 7. The output region of the Trojan-free AES circuit. The box labeled rf represents an instantiation of the custom module final_round, which is defined in the Verilog file round.v. This module represents the tenth round of the AES encryption process. This round is similar to previous rounds. It accepts the last round key and the last intermediate state and produces the circuit's final output...24
Figure 8. The internal functionality of the Trojan-free AES circuit. The expand_key_128 modules generate intermediate round keys. Note that out_1 and out_2 hold the same value, so each round key is derived directly from the previous key. 25
Figure 9. The structure of the benchmark AES-T100. Note the absence of the Trojan_Trigger module that was shown in the typical Trojan layout. No module is necessary to represent an always-on trigger. 26
Figure 10. Detail view of the TSC module of AES-T700. The output load is composed of 8 bits, each repeated 8 times. These bits are the result of XOR operations between key[7:0] and lfsr[7:0]. To simplify this view, we have hidden the XOR gates load0_i__0 to load0_i__5. These XOR gates function in parallel to the two shown, with each gate operating on a different bit pairing. 27
Figure 11. One shift operation of the lfsr module. The value 0x99999 is the initial value of the AES-T700 LFSR register. The new high-order bit is calculated from an XOR of bits from positions 3, 7, 11, and 15. After this calculation, bits are shifted to the right, and the XOR result is used to fill in the missing bit. 28
Figure 12. Detail view of the Trojan in AES-T200. Note the addition of the input data to the lfsr_counter module. The value of data is determined by state, one of the inputs to the overall circuit. 30
Figure 13. AES-T300 modifies the AES_128 module by adding eight additional outputs. These outputs carry eight of the round keys used during the AES encryption process. Note that the last two round keys are not leaked, but we assume that the attacker either knows or has the ability to discover the mechanism used to generate the key. 32
Figure 14. Detail view of a shift register from AES-T300. Note that the world view shown here is a partial schematic, depicting only 1 of the 8 registers. On a reset signal, the register is fed with an initial value of 10101010_2. Afterwards, register's value remains unchanged unless the input G is 1_2. G is the result of AND and XOR operations performed using state and the first round key. 33
Figure 15. A waveform diagram displaying the actions of the shift registers in AES-T300. The top waveform displays the actions of the system clock. One

step of rotation for a register means a transition from a value of 0xAA to 0x55 or back. In the view shown, each round key has a different value, so the registers are rotating at different times. For example, SHReg6 rotates every time the clock changes because it satisfies the AND and XOR test for the entire period shown. 34
Figure 16. The full view of AES-T400. Note that this benchmark follows the typical structure displayed earlier. The functionality module is named AM_Transmission, but it accepts inputs from the wires key and Tj_trig. Like the TSC module present in other benchmarks, AM_Transmission operates without disrupting the core functionality of the aes_128 module...35
Figure 17. A detail view of the AES-T400 trigger. Detected_i represents the actual comparison of the incoming value state against the predefined value of 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. Note that due to features of AES-T400's functionality, additional gates have been added to set Tj_Trig to 0_2 within two clock cycles of activation. 36
Figure 18. The shift register of AES-T400. When Tj_Trig = 1_2, this register is loaded with key. Every time that the register's C input is set to 1_2, the register input is updated with the output from SHIFTReg0_i. This output is the register's previous value with every bit shifted to the right........39
Figure 19. The apparent structure of AES-T500. Note that while no trigger module has been explicitly defined, the TSC module contains logic dedicated to the purpose of triggering the Trojan after specific inputs have been observed. Also note that Vivado has not elaborated on the contents of the module TSC. Vivado takes this approach to modules that do not have a specific output. To work around this, we added an output to the TSC module, directly using an existing register to provide the output's value. With this change, we were able to generate detail views of the internal workings of the TSC module. 40
Figure 20. A detail view of the first two comparisons performed by the trigger in AES-T500. Vivado represents the first comparison as the RTL_ROM module State0_i. This module is slightly overshadowed by the long binary input to State1_i. Note the result of this comparison is stored in the RTL_LATCH State0_reg. The second comparison is depicted as a direct comparison of state to a given value using an RTL_EQ module...........42
Figure 21. Detail view of the Trojan functionality of AES-T500. DynamicPower_reg is initialized with an alternating pattern of $1010..._2$. The wire feeding into G is Tj_trig. When this wire has a value of 1_2, the register will rotate on every clock cycle. The bus from Q to D represents the altered value being fed back into the register. Note that this view cuts off the extra output we added to force Vivado to generate this image. 43

Figure 22. Leakage from the AES-T600 Trojan. Note that the wires INV1_out through INV11_out are defined as wires, but not circuit outputs in the Trust-Hub code. According to the documentation for this benchmark, the leaked bit can be recovered by measuring the leakage current. To produce this schematic, we extended the wires into module outputs. Without the change, Vivado would refuse to expand the TSC module. 44
Figure 23. Detail view of the Trojan_Trigger module of AES-T700. Activation is represented by a 1_2 signal on the output Tj_Trig. This signal begins when the correct input state is observed, as compared to the value in Tj_Trig_i. A 1_2 signal on rst will deactivate the Trojan by acting as the "clear" input of the RTL_LATCH. 46
Figure 24. Detail view of the TSC module of AES-T700. Note the presence of the Tj_Trig input. This input only applies to lfsr, meaning that only the operation of that module is affected by the activation of the Trojan. 47
Figure 25. The counter in AES-T900. Vivado is not representing this counter using a register module. Instead, the value of the counter is represented by the loop passing between Counter0_i and Counter_i. Note that the select bit of Counter_i is provided by rst. 49
Figure 26. Detail view of the TSC module in AES-T1000. Note that this module combines features from AES-T200 and AES-T700. The LFSR register is initially fed from data, which is state in the overall circuit. Tj_Trig is used to control only the rotation of the LFSR. 50
Figure 27. Inputs of the basicRSA circuit. The key inputs to be aware of are inExp and indata. Depending on the exact usage case of the circuit, at least one of these values is secret information not meant to be shared with the outside world. As a result, the adversary would attack confidentiality by causing the circuit to leak one or the other of these values. 59
Figure 28. The outputs from the basicRSA circuit. Output cypher carries the result of the RSA operation. Due to the structure of the algorithm, this can either be an encrypted ciphertext or a decrypted plaintext. Output ready notifies surrounding architecture that an RSA operation has been completed. 60
Figure 29. The complete malicious inclusion of basicRSA-T100. Comparator eqOp_i compares the input indata with 0x44444444. The result of this operation is used as the select bit for the mux gate cypher_i. The input I1 of cypher_i is the bus product, which carries the result of iterative modmult operations. When the RSA algorithm has finished, product carries the correct final result. Input I0 of cypher_i is linked directly to circuit inExp. The output of cypher_i feeds directly to output cypher. 61
Figure 30. The complete Trojan in basicRSA-T200. Like basicRSA-T100, this benchmark triggers on a simple comparison operation. InputExponent_reg is added as an intermediate storage of inExp

prior to the use of that input for the main RSA operation. If the Trojan is triggered, this intermediate storage location allows the value of inExp to be substituted with 0x00000001. The basicRSA circuit will then conduct encryption or decryption operations using the substitute value. 62

Figure 31. The trigger mechanism in basicRSA-T300. The core counter is composed of register TrojanCounter_reg and adder plusOp_i. Note that Q value of this register loops back through the adder, which adds 1 to the value on every loop. Also note that the register will only update its value if the Q value of TjEnable_reg is 1_2..64

Figure 32. Circuit diagram of the functionality of basicRSA-T400. The comparator ItOp_i is the last component of the counter-based trigger mechanism. The result of this comparison will be used to select between using inExp to perform the RSA operation, or replacing it with the adversary's chosen exponent: 0x009ADD0A. 66

Figure 33. High level schematic of the Trojan-free version of RS232. Note that the modules iRECEIVER and iXMIT operate in near isolation. They share the same clock and reset signals, but otherwise, each has its own inputs and outputs. Module iXMIT converts the byte xmit_dataH into a series of 1-bit signals transmitted through uart_XMIT_dataH. Module iRECEIVER does the opposite, accepting a series of bits from uart_REC_dataH and converting them to the byte rec_dataH.............69

Figure 34. A diagram of the state machine in the iRECEIVER module of the RS232 circuit. Each time the machine leaves the r_SAMPLE state, a single bit is read from uart_dataH and added to the output register. State r_WAIT and register bitCellCntrH are used to control timing between each read operation. States r_START and r_CENTER confirm the initial 0_2 bit that signals the start of the message. The transition to r_STOP occurs after 8 bits have been read and another 15 clock cycles have passed. This extra time allows the circuit to account for the final bit of the serial message. 70

Figure 35. Partial schematic showing the trigger registers of RS232-T100. These registers, in conjunction with the current value of rec_dataH, form the 19-bit trigger value of this Trojan. AND gates are used to funnel these inputs into a single result wire labelled ena. The world view above highlights the registers, showing how Vivado places them in the schematic...73

Figure 36. Schematic of the gates controlling the functionality of RS232-T100. The AND gate ena_i is the last gate in the trigger comparison process. The result of this operation is used as the select bit for two mux gates. One gate determines the output rec_readyH, and the other determines the output rec_dataH. Note that the result of the mux gate rec_dataH_i is fed back to another area of the receiver sub-circuit. This allows it to be used as part of the triggering conditional..74

Figure 37. The Trojan in RS232-T200. The three gates shown make up a 10-bit counter, which increments every cycle while the Trojan is active. Note that this counter is not used as an input to any other part of the circuit. Also note that output count_l was artificially added to prevent Vivado's automatic optimization from excising both the Trojan and the trigger. 76
Figure 38. Partial Schematic of the trigger mechanism in RS232-T300. Register count_in_reg, the adder, and the mux gate count_in_l form a counter that is designed to count from 0 to 0xFFFFFFFF. When the counter reaches the final value, the ROM unit DataSend_ena_reg will send a 1_2 signal, representing activation. 77
Figure 39. The trigger mechanism of RS232-T400. The RTL_EQ primitive is responsible for comparing rec_dataH against xmit_dataH. The result of this operation is fed into ROM cntr_i. If the values are equal, the value of register cntr_reg will be set to 1_2. This wire is used as a select bit for a mux gate, which controls the final rec_dataH output from the circuit...79
Figure 40. The Trojan functionality of RS232-T400. The key feature of this Trojan is the mux gate rec_dataH_temp_i, which is used to determine whether the final circuit output should be the correct value of rec_dataH, as determined by the iRECEIVER module, or a reordered combination of bits. Note that both iRECEIVER and the overall circuit have an output labelled rec_dataH. In every other circuit in this group, the distinction is unnecessary because the module output is fed to the overall circuit output without modification..80
Figure 41. The RTL layout of the state machine that controls the trigger of RS232-T600. Each state_DataSend_i__# ROM module represents a potential state. The output of that module is dependent on the current value of xmit_dataH. The mux gate state_DataSend_i__4 will only select one of these values to pass through to state_DataSend_reg. This register actually contains the select bits responsible for that choice; the current state is responsible for determining which values can potentially be passed to the register. 82
Figure 42. A partial schematic representing a typical layout for the inclusion in the gate-level RS232 circuits. This particular schematic was generated from RS232-T1000. Note that while each of the structures shown here is depicted as a custom module, the labels AND2X1, OR4X1, etc…, reveal them to be implementations of common logic gates. This is a result of uart.v using gates defined in a non-standard Vivado library. 89
Figure 43. Partial schematic of the inputs to U296. Remember that we have previously established that the output and the inputs of U296 must each have a 0_2 value in order to trigger the Trojan in this circuit. The NAND truth table shows the only combination of inputs that will yield 0_2 as an output. 91

Figure 44. Partial schematic of the inputs to U294. Note that the value of iXMIT_bitCell_cntrH_reg_2_(QN) is being reused. Many of the circuits reuse source values and other gates in the trigger mechanism. This can lead to contradictions like the one discussed here....................................92
Figure 45. Schematic of the functional portion of RS232-T1000. Note that U303, U304 and U305 are all AND gates. All three of these gates share the common input iCTRL. Module U303 directly controls the circuit output xmit_doneH, and U305 is only separated from uart_XMIT_dataH by a single intermediate module. The internal functionality of this OR-AND-invert (OAI) module is shown in the insert above U3. Module U304 does not affect the logical operation of the circuit..94
Figure 46. Partial schematic of U296 and inputs relevant to this discussion. U293 and U294 are both NAND gates with required outputs of 0_2. As a result, all of their inputs must be 1_2. This includes the QN outputs from iXMIT_state. 95
Figure 47. Partial schematic showing a trace from U88. Of particular interest is the module U91. In order to add clarity to later discussions regarding this module, the internal structure is shown here. The truth tables shown here indicate the possible input combinations at each stage of the diagram. Note that only one of the inputs to the AND gate must be 0_2, and that the other can be 1_2 or 0_2 without affecting the final output. 96
Figure 48. Partial schematic of U292 and its inputs. The AOI truth table describes the possible inputs to this module. Additional tables are used to illustrate the required values at each intermediate stage of the module. 97
Figure 49. As in RS232-T1000, the Trojan functionality is determined by iCTRL's influence over an AND gate. In this case, there is only the single AND gate U305, which influences, but does not directly control the transmission output uart_XMIT_dataH. 98
Figure 50. Partial schematic showing the trigger of RS232-T1200. For space considerations, the inputs of the NAND gates have been omitted, but all 6 are 4-input gates. Note that, to produce an output of iCTRL = 0_2, the outputs of each of these NAND gates must be 0_2, as indicated by the OR truth table. 100
Figure 51. Schematic for the trace of inputs from NAND gate U292. This gate directly accepts the Q outputs of iXMIT_state_reg_0_, iXMIT_state_reg_1_, and iXMIT_state_reg_2_. We will accept these values as source values. 101
Figure 52. Schematic of the source values leading to U294. While no flip-flops directly provide inputs to U294, the AND gate U90 still provides a clear value for iXMIT_bitCell_cntrH_reg_0_. Using this value and the properties of U211, U215, U217, and U218, we can determine the requirements for iXMIT_bitCell_cntrH_reg_1_ and iXMIT_bitCell_cntrH_reg_3_. 102

Figure 53. Schematic of the source values leading to U293. The flip-flops shown here correspond to the RTL register bitCountH. Note that from this schematic, we can determine a relationship between iXMIT_bitCountH_reg_1_(Q) and iXMIT_bitCountH_reg_0_(Q), but we cannot assign precise values to them. 104
Figure 54. Schematic of the source values leading to U295. Note that the A input of U295 is provided by U216(Y), which we discussed in our examination of U294. This reuse of source gates is common among the gate-level RS232 circuits. 106
Figure 55. Partial Schematic of the source values for U91. Note that the values for every flip-flop shown here have already been determined. We will use this diagram for verification purposes. 108
Figure 56. Partial schematic of the inputs of U87. As with U91, we will use known values to demonstrate that RS232-T1200 does not contain a contradiction. 109
Figure 57. Schematic of the source values leading to U297. Note that U297 can be used to directly determine the iRECEVER_state values, while iRECEIVER_bitCell_cntrH requires several stages of analysis. However, there are no ambiguous input combinations in this set. Each gate's required output allows for only one possible combination of inputs. 110
Figure 58. Schematic of the source values leading to U300. Note that all of these values are Q outputs from inserted flip-flop modules. 112
Figure 59. Schematic of the source values leading to iDataSend_reg_1. Note that SN is set to the constant 1_2, meaning that input combinations in the bottom most rows of the SDFFSR truth table will never be observed. 113
Figure 60. The Trojan functionality of RS232-T1200. Note that the value of gate U303 is used as the output xmit_doneH. This allows the iCTRL wire to force that output to 0_2 after the Trojan is triggered. 116
Figure 61. Partial schematic of the trigger for RS232-T1300. This Trojan has a slightly different structure than that used in other benchmarks in this group. The OR gate U301 has been removed from the structure, and NAND U297 is directly connected to U302 118
Figure 62. The functionality of RS232-T1300. Note that U304 controls the rec_readyH output, and U303 controls xmit_doneH. Since both are AND gates, the iCTRL wire can be used to force these outputs to 0_2 119
Figure 63. Partial schematic of the trigger mechanism of RS232-T1400. Only the gates that display the contradictory requirements are shown here. The Y output of U302 is iCTRL, which represents the activation state of the Trojan. For the Trojan to be active, this output must have a value of 0_2. Truth tables have been provided for the relevant gates. The highlighted entry in each truth table represents the output required to activate the Trojan. 120

Figure 64. Partial schematic of the trigger mechanism for the RS232-T1700 benchmark. This schematic illustrates the new ena input wire that was added as part of this inclusion. In this benchmark, ena is used as an input to each of the NAND gates in the inclusion. This replaces some of the intermediate inputs that were used to control the Trojan in other benchmarks in this set. 124
Figure 65. The complete inclusion in RS232-T1800. Note that the output of the final INV gate is not used anywhere else in the circuit. The Trojan is almost completely isolated from the rest of the circuit, sharing only the sys_clk input....125
Figure 66. Schematic showing the inclusion in RS232-T1900. Note that U302 has been replaced by U296. In addition, the circuit is dependent on 1_2 values on ena and the iDatasend flip-flops. 126
Figure 67. A portion of the Trojan-free AES circuit diagram. We will use this to identify specific internal wires and busses that can be used as inputs to a Trojan_trigger module or outputs form a TSC module, similar to those used in earlier AES benchmarks....133

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Table of register values for RS232-T1200. This table represents grouped flip-flop Q values required to activate the Trojan in RS232-T1200. We have presented the flip-flops in this fashion so that we can also present the value of the RTL register, which is useful in explaining the purpose of each flip-flop group......115

Table 2. Table of source values for the RS232-T1300 trigger mechanism. The flip-flops have been grouped into logical registers, to demonstrate the full value of each group. Note that iXMIT_xmit_ShiftRegH_reg is actually an 8-bit register, but that bits 0 through 4 have no impact of the Trojan trigger. Also note that the outputs iXMIT_bitCount_reg_1_(Q) and iXMIT_bitCount_reg_0_(Q) must have opposite values, but that the order of those values does not alter the result of the Trojan trigger. 117

Table 3. Table of triggering inputs for RS232-T1600. Note that inclusion in this circuit does not link directly to the XMIT_state_reg flip-flops. These source values are now fed to the Trojan by way of XOR gates, an approach that adds flexibility to their required values. 122

Table 4. Table of flip-flop Q values required to trigger the Trojan in RS232-T1700. Note that this table does not include iRECEIVER_bitCell_cntrH. The flip-flops for that register are not found in the structure of RS232-T1700's trigger......123

Table 5. Table of flip-flop Q values required to trigger the Trojan in RS232-T1900. Note that the iRECEIVER conditions have all been removed from this inclusion. 126

Table 6. Triggering inputs for RS232-T2000. Note that the receiver-side logic has been completely removed from this inclusion's trigger. Instead, the B input of U302 is provided by the flip-flop iDatasend_reg 127

Table 7. Table listing the source values required to set iDatasend(Q) = 02. Wires sys_clk, ena, sys_rst, and test_se are circuit inputs. The other values identified here are flip-flop outputs......128

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3PIP	3rd party intellectual property
AOI	AND-OR-invert
ASIC	application-specific integrated circuit
COTS	commercial off-the-shelf
DEF	design exchange file
DSP	digital signal processor
FPGA	field-programmable gate array
HDL	hardware descriptor language
HOST	hardware oriented security and trust
IC	integrated circuit
IMP	Illinois malicious processor
LFSR	linear feedback shift register
OAI	OR-AND-invert
RTL	register-transfer level
XOR	exclusive or

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Drs. Salmani and Tehranipoor for assembling the benchmark collection discussed in this thesis. This seems to be the first serious attempt at a publically available benchmark collection for hardware Trojans, and I believe that it will be very helpful to future researchers. Dr. Salmani also took time to answer questions that I had about the features of many benchmarks discussed in this thesis.

I would also like to thank Dr. Ryan Kastner, Mr. Janarbek Matai, and Dr. Wei ("Vinnie") Hu of University of California, San Diego, for their technical assistance. In particular, these researchers made themselves available to answer questions about advanced features of the Vivado software. Their assistance made it possible to synthesize the benchmarks and subject them to simulation.

I would like to thank Mark Gondree for serving as my second reader and helping me to clarify the contribution of this thesis.

Finally, I would like to thank my advisor, Professor Ted Huffmire, who helped me to plan and organize this thesis and assisted me by providing lab space for me to conduct this research. Without Professor Huffmire's aid, this thesis' contribution to HOST research would be substantially reduced.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND MOTIVATION

The Department of Defense (DOD) is becoming increasingly dependent on untrusted integrated circuits (ICs). The Pentagon annually spends $3.5 billion on ICs destined for use in military equipment [1]. A single military plane may be constructed with over 1000 circuits [2]. If one of these circuits should fail during flight, the plane might lose control or communications from the aircraft may be broadcast without any scrambling or encryption. Such a failure could be the result of a deliberate hardware modification known as a hardware Trojan.

A hardware Trojan, also referred to as a malicious inclusion, is a deliberate alteration to a piece of electronic hardware that causes that device, under certain conditions, to display undocumented functionality. Hardware Trojans may be added to varying items of hardware, including application-specific integrated circuits (ASICs), digital signal processors (DSPs), microprocessors, and other commercial off-the-shelf (COTS) products [3]. These alterations may also be applied to field-programmable gate arrays (FPGAs), either as changes to the underlying device or as subverted bitstreams meant to change the hardware configuration [3].

The Trojan functionality, referred to as the “payload,” may disable some part of the circuit, transmit information to the adversary, or overwrite output values from the circuit [4]. The activation condition, referred to as the “trigger,” may be a specific combination of inputs, multiple input combinations in a predefined order, or the passage of a set amount of time [5]. A trigger can also be constructed from more than one of these elements. For example, a circuit may ignore the triggering combination until after a certain number of operations have been completed [5]. Note that it is also possible to create a Trojan with an “always-on” trigger. In these designs, the functionality operates at all times. Several of the benchmarks discussed in this thesis use this model.

A malicious insider can insert a hardware Trojan at any stage of the design process, using a variety of techniques [4]. A Trojan to reduce a circuit’s reliability can be implemented simply by changing the geometry of a single wire [3]. More complex

Trojans require several thousand additional transistors, but those transistors have been added to a circuit that contains billions of other transistors [2].

According to [6], the adversary can also compromise circuits by altering the components of the underlying design. Today, many chips are designed at least partly overseas, either because the company has purchased 3PIP cores, or because the designer has acquired at least some of the HDL in their design from searching online forums [6], [2]. A Trojan in any one of these cores can disrupt the operation of the entire device [6].

Testing for a hardware Trojan is difficult. As we have already discussed, a well-designed Trojan uses only a small fraction of the physical structure of the overall circuit. In a physical examination, a tester would need to view billions of transistors in the circuit, and find as few as 1000 that had been added. Note that a thorough physical examination is destructive to the part under examination, because each layer of transistors must be ground away to reveal the layer underneath [2]. Further, the examination is of limited utility. Manufacturers use this technique on a single chip from a batch of thousands, based on the assumption that the manufacturing process will produce identical parts. However, Adee points out that a malicious insider can replace the circuit mask for a single silicon wafer, resulting in a Trojan that has been inserted into that chip alone. Even if the tested part is proven to be free of Trojan functionality, the status of the other parts in the bath remains unknown [2].

As Adee explains in [2], discovering a hardware Trojan with functional testing is also difficult. To find a discrepancy between a circuit input and the expected output, it is necessary to actually trigger Trojan functionality. For a simple combinatorial Trojan, a tester would need to apply each possible input combination in turn until either the Trojan is triggered or every possible input has been tried. For a Trojan with a 512-bit input, there are more than $13.4*10^{153}$ combinations.

Hardware Trojans have become a particular risk due to the rise of global manufacturing. Many IC designers, including Sony and LSI Corp, have stopped producing their own ICs. Completed designs are sent to dedicated foundries, owned by

manufacturing companies. Very few of these foundries are located in the United States, and a growing number are located in China [2].

Acknowledging the potential dangers from hardware Trojans, the DOD has established a “Trusted Foundry” program, which certifies that certain IC production facilities are trusted to not alter the functionality of circuits they manufacture [1]. As of 2009, only 2% of ICs purchased by the Pentagon came from foundries certified under this program [7].

In this thesis, we examine an existing, public, collection of hardware Trojans. Dr. Hassan Salmani and Dr. Mohammed Tehranipoor have created 92 hardware Trojans and posted them as benchmarks at the website trust-hub.org [8]. To the best of our knowledge, this is the only publically available set of hardware Trojan benchmarks. Other researchers have implemented their own hardware Trojans and used them in experiments, but we have been unable to find any means to access the HDL or bitstreams for any of these circuits. The Trust-hub benchmarks are the only hardware Trojan circuits that are available to be implemented by any researcher. This availability allows the Trust-hub benchmarks to serve as a common standard for measuring the effectiveness of hardware Trojan detection and mitigation techniques.

We conduct a thorough examination of 46 of the benchmarks from the Trust-hub collection. The purpose of this study is to determine the challenges that must be overcome in attempting to synthesize these benchmarks and conduct simulated experiments using them. To assist future researchers in conducting studies using these benchmarks, we have provided the following supplements to the existing documentation and resources:

- We document the procedure for establishing a simulation environment using the Xilinx Vivado design suite. We also document the procedure for creating a Vivado project from one of the provided benchmarks. To complete our general discussion of Vivado, we describe the process of importing a test bench and conducting simulation of circuit activity. Note that some benchmarks discussed here will either fail to synthesize as

written, or produce simulation results that do not agree with the provided documentation. To assist future researchers, we document changes we have made to the provided HDL in order to complete synthesis and simulation. We have also written and provided two library files, which are necessary in order to synthesize the gate-level RS232 benchmarks. These libraries are included in a software archive provided in conjunction with this thesis. This archive is discussed in more detail in the appendix.

- For each of the 46 benchmarks discussed in this thesis, we provide a detailed description of the trigger and functionality of the hardware Trojan that has been added to the circuit. We also identify benchmarks that provide incomplete or inaccurate documentation and provide corrections where appropriate. In particular, some of the RS232-based benchmarks provide documentation that either fails to identify the triggering condition for the included Trojan or incorrectly identifies that condition. For each of these benchmarks, we describe the correct triggering combination.
- Finally, we have provided three test benches for use with the RS232 benchmarks. These test benches, which have been included in the thesis software archive, can be used as baseline models for more complex test benches and simulations. Note that Salmani et al. already provide test benches appropriate for use with the AES and basicRSA benchmarks. These are discussed in the appropriate sections of this thesis.

This thesis also aims to present a Computer Science perspective on hardware-oriented security and trust (HOST) research. This perspective is essential to understanding how malicious inclusions inserted at the digital logic level cause security failures at higher levels of abstraction.

Finally, this thesis recommends malicious inclusions that could be added to the benchmark suite. A high level sketch is provided for an inclusion that is complementary to the inclusions present in the suite, but that allows researchers to examine features not already demonstrated by existing benchmarks.

In a broader context, this work will advance Computer Science research and education by providing Computer Science students with the means to contribute meaningfully to HOST research. For example, the techniques described in this thesis could aid teachers in designing lab exercises for a hardware security course for Computer Scientists. Students with limited prior exposure to this material will benefit from the description of the pitfalls and difficulties involved in using Xilinx tools to simulate the hardware Trojan benchmarks available on the Trust-Hub website, or to assemble experiments based on other devices.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RELATED WORK

Wang et al. developed a taxonomy for hardware Trojans in [3]. This taxonomy classifies Trojans according to physical structure, trigger mechanism, and payload actions. These researchers also provide a preliminary list of applications that are vulnerable to hardware Trojans. Many Trojan implementations and detection methods reference this work. In [7], Banga and Hsiao provide additional terminology that is relevant to this work. They define a combinatorial Trojan as a Trojan that triggers on a specific combination of input values, and a sequential Trojan as a Trojan that triggers after seeing several combinations in some specific order.

According to [3], the preferred hardware Trojan detection techniques are automatic test-pattern generation (ATPG), failure-based analysis, and side channel analysis. ATPG designs a sequence of inputs based on the circuit's netlist, and compares the resulting outputs from the expected outputs. This technique is ineffective against Trojans that modify the circuits logical layout. Failure based analysis uses microscopic imaging techniques and voltage induction to verify that a circuit conforms to its specified design. Most forms of failure based analysis are destructive to the chip they are performed on. Like ATPG, side channel analysis uses a series of input values to engage the circuit, but instead of concentrating on digital output values, side channel analysis measures analog current at intermediate stages of circuit logic, detecting hardware Trojans through unexpected activity in the circuit. Trojans may also be written to use side-channel analysis as a tool to leak information to malicious insiders.

All three of these techniques operate only after fabrication. If a design-phase Trojan is detected, then all chips produced from that design are compromised and will need to be replaced. The chip design must be corrected, and the new design must be prototyped and tested before a new batch can be manufactured.

A number of researchers have provided detailed evaluations of these detection techniques or added refinements to increase the success rate of hardware Trojan detection. Kutzner et al. conducted a trial of IC fingerprinting and side channel analysis

techniques. In this analysis, the researchers determined that the presence of operational noise noticeably reduced the reliability of these techniques [9]. Banga and Hsaio used side channel analysis on localized regions of a circuit to increase the reliability of Trojan detection [7].

In [6], Zhang and Tehranipoor used a mixture of formal verification, code coverage and ATPG to increase confidence in the presence or absence of hardware Trojans in IP cores provided by third parties. The researchers apply their technique to the RS232 series of hardware Trojans from [8]. These Trojan circuits will be discussed in detail in Chapter 4 of this work.

In [10], Alkabani and Koushanfar propose the use of the “consistency” metric in side channel analysis. Using this metric, the researchers are able to evaluate the presence of a Trojan by repeated measurement and comparison of current leakage from the circuit.

In [11], McIntyre et al. devised a Trojan detection system that allows gradual redefinition of circuit trust in a multi-core system by comparison of results from different cores. Since Trojans commonly trigger on specific input sequences, the researchers devised a method for reordering subtask operations without changing the final result. Two different, but equivalent sequences are provided to two different cores. If both return the same result, the system has high confidence that the sequences did not trigger the same Trojan, and that the specific operation was completed without Trojan interference. If the circuits return different results, the system uses additional subtask orderings and cores to determine which core is producing an invalid result, and is therefore likely to contain a malicious inclusion.

In [12], Li and Lach propose another long-term, post-fabrication mechanism for detecting hardware Trojans. Here, the researchers state that the activation of the Trojan through its predefined trigger mechanism will cause an immediate, noticeable change in circuit delay characteristics. Continuous measurement of signal propagation allows the researchers to observe the change in delay and designate the circuit for further investigation. The technique is vulnerable to false positives caused by changes in

temperature and voltage, and sufficiently fast-acting Trojans may not cause a noticeable delay impact.

In [13], Hicks et al. have developed Unused Circuit Identification (UCI), a design-phase test that identifies gates and paths that do not change signals during expected input sequences. Based on the fact that malicious inclusions are designed to not activate during normal circuit operation, the researchers remove UCI-selected logic from the circuit. In case UCI removes valid, non-malicious circuitry, these researchers also implemented a technique for adding software to simulate the activity of removed logic.

Other researchers have created sample hardware Trojans, demonstrating techniques that are available to adversaries. In [14], King et al. implemented the Illinois malicious processor (IMP), a general-purpose hardware Trojan. The inclusions in the IMP are designed to support more complex attacks from software that can be implemented against a Linux system installed on the IMP. These researchers implemented a login backdoor, a password-stealing mechanism, and a privilege escalation attack as proof-of-concept. Dr. Salmani and Dr. Tehranipoor have designed a variety of sample hardware Trojans and posted them at the Trust-Hub website [8]. This thesis will analyze a selection of those sample Trojans, detailing their malicious functionality and outlining the processes needed to simulate and study the malicious activity.

THIS PAGE INTENTIONALLY LEFT BLANK

III. METHODOLOGY

This chapter will describe the methodology used in analysis of the available Trojan benchmarks.

A. GETTING THE BENCHMARKS

The Trojans were all downloaded from [8]. Benchmarks on this site can be categorized by the circuit they modify. For example, there are 21 Trojans designed to undermine the effectiveness of a 128-bit AES encryption circuit. These circuits are labelled `AES-T100` through `AES-T2100`. The naming convention includes the final two zeroes as a means to distinguish between different placements and versions of the same Trojan in a particular circuit. For example, `RS232-T901` is a slightly altered version of `RS232-T900`.

To download a Benchmark, navigate to [8]. Select the name of the desired Trojan from the "Resources" column. A partial description of the Trojan will appear under "Info," along with two links. Figure 1 demonstrates the relevant portion of the benchmark selection process. The full description can be read by following the "Learn More" link provided.

Select the download link to acquire the Benchmark archive. Each archive contains source files that define the Benchmark. For most Benchmarks in this collection, these source files are HDL code written in Verilog or VHDL. Several Trojans, such as the EthernetMAC10GE series, include only DEF files, which represent the physical layout of the gates and wires in the circuit. This thesis does not discuss the operation of Benchmarks composed of DEF files.

The archive will also contain a PDF file and a README. These documents both contain the same description that can be read by following the "Learn More" link. They may also provide additional information about the benchmarks, including the results of tests run by Salmani et al.

The archive may contain source code for a Trojan-free version of the benchmark or a test bench designed to demonstrate the function of the circuit.

Figure 1. Selection of a resource from the Trust Hub site. The button labeled "Download (ZIP)" will directly download an archive file containing all of the resources provided by Salmani et al. for the selected Trojan. Note that some Trojans are stored as RAR archives. The "Learn More" link will lead to a dedicated page for the selected Trojan. Note that the b19 Trojans are stored as multi-part archives, and it is necessary to visit the dedicated page and download parts from the "supporting files" section.

B. SOURCE CODE ANALYSIS

Most of the provided circuits included source code written in Verilog or VHDL, so we began by attempting to perform a static source-code analysis of the Trojans. We used MD5 hashes to quickly find duplicate source code files. If a file's hash matched that of a file from the Trojan-free implementation of the circuit, we could safely assume that the HDL defining the Trojan was not present in that file. If two files from different benchmarks produced the same hash, then we would only need to analyze that specific version of the file once. Any insights gained from that analysis could be applied to our analysis of other benchmarks that contain the same version of that file. After duplicate files were eliminated from the analysis group, we subjected the remaining files to `diff` tools to find the exact differences.

Some of these files differed only in the positioning of whitespace, which does not influence the functionality of circuits synthesized from Verilog or VHDL. The files from the RTL-based RS232 benchmarks commonly displayed this trait. Some of the remaining files drastically reordered HDL instructions, but did not alter the logic of those instructions. We recorded the list of logically equivalent files, then removed them from the analysis group.

The remaining files included small sections of extra logic, which we were able to evaluate more closely. In most cases, we were able to confirm that the extra logic served to define the malicious inclusion. Examining this smaller portion of source code allowed us to more easily specific inputs or registers that fed into the trigger mechanism and outputs that were controlled by the Trojan functionality.

C. SETTING UP THE ENVIRONMENT

The second stage of analysis involved synthesis and error checking. We established a dedicated test environment for the remainder of the analysis process. On a Windows 7 virtual machine, we installed Xilinx Vivado Webpack version 2013.4.

To download Vivado Webpack, you first need to establish an account on the Xilinx website. New user accounts can be created at `https://secure.xilinx.com/webreg/createUser.do?` You will need to provide your name, a user ID and password, and an email address from a university or business. The site will confirm your registration using the email you provide.

Once you establish an account, use a browser to navigate to `http://www.xilinx.com/support/download.html.` Select your preferred version of the software from the column on the left side of the page. Select the link for "All OS Vivado and SDK Full Installer." This link will download a 6 GB TAR/GZIP archive. Extract this file to a folder of your choosing. The total size of the extracted archive should be roughly 6.8 GB. The installer should be `xsetup.exe`, located in the top level of this directory.

D. SYNTHESIZING AND VIEWING A CIRCUIT

We imported the source code of each Benchmark as a separate RTL project. We also created projects for the Trojan-free versions of circuits. To do this, start Vivado and select the "create a new project" link on the opening screen. This will open the new project wizard, which provides some guidance when creating a new project. The new project wizard consists of eight dialog windows, as follows.

- **Create a new Vivado Project**: This is a short introduction to the new project design process. Select "next."
- **Project Name**: This dialog allows you to name your project and select a folder for the project directory. Note that you can specify the full path yourself, or allow Vivado to create a directory with the same name as the project.
- **Project Type**: For all of the Benchmarks discussed here, we selected "RTL Project."
- **Add Sources**: Most of our work was done here. Choose "add files" to open a file chooser dialog box. For convenience, Vivado supports selecting multiple files in a single file-chooser instance. Navigate to the source directories of a downloaded benchmark and select the HDL files there. These may be Verilog files, with a ."v" extension or VHDL files with a ."vhd" extension. Do not add test benches at this time. Using test benches as design sources produces strange results from synthesis, schematics and simulation. Before selecting "next," ensure that "Copy Sources into Project" is checked.
- **Add existing IP**: This window allows you to add 3PIP cores to your project. None of the projects discussed here require any files to be added to this section.
- **Add Constraints**: This window allows you to add simulation and synthesis constraints to the project. These constraints can be used to control timing and gate placement in a circuit. The Benchmark groups that we demonstrate in this work do not contain constraint files.

- **Default Part**: Your selection in this window is dependent on what physical hardware, if any, you expect to use for physical demonstration. All of our analysis in this work was conducted within Vivado. We selected the Artix-7 FPGA under “boards,” since we knew our advisor had at least one FPGA within that family.
- **New Project Summary**: This final dialog summarizes all of your previous selctions in a single window. After reviewing them, select “Finish,” and Vivado will construct this project from the files you have provided.

Once a project has been successfully created or opened, you will have access to Vivado’s main screen, as shown in Figure 2. Most functions that you will need during the simulation of these benchmarks can be found in the left pane, labelled “Flow Navigator.” My first action after creating a project was to select the “Run Synthesis” instruction from this pane. For most of the benchmarks discussed in this thesis, the default synthesis settings are appropriate.

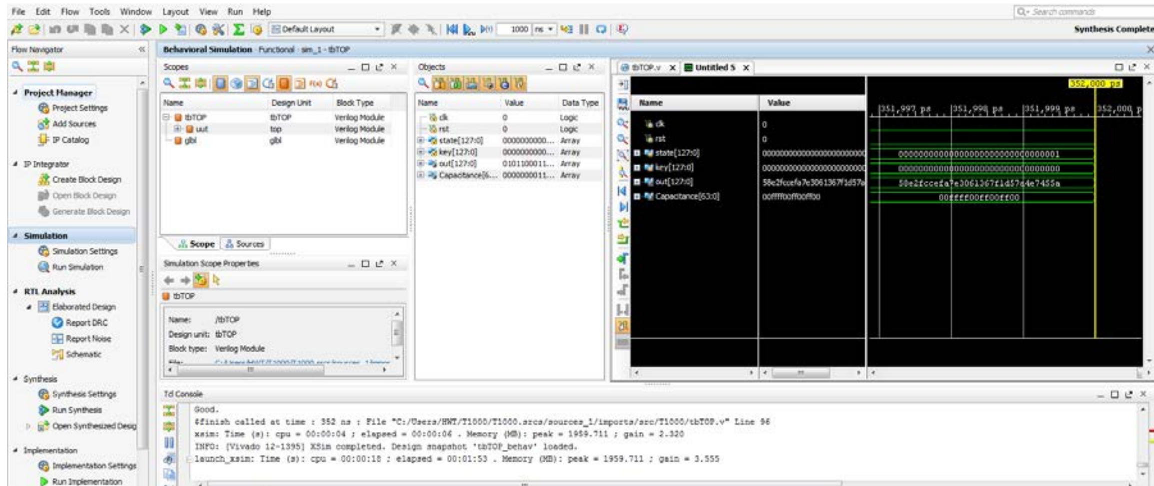


Figure 2. Layout of the Vivado Main Screen. Synthesis, Simulation and schematics are opened from the flow navigator on the left. The window on the far right shows the results of these commands. In this case, the window shows a waveform diagram resulting from simulation. Note that the results window can be popped out and viewed separately from Vivado’s main window. The central windows allow some customization of the results window display. Errors and other messages are reported in the bottom window.

E. VIEWING SCHEMATICS

After synthesis, we were able to examine the provided benchmarks by means of Vivado's schematics generation. There are two schematic generation mechanisms available. Generating a schematic from a synthesized design will produce a gate-level schematic, treating each 1-bit wire as a separate entity. A 64-bit register would be displayed as 64 flip-flop modules. Generating a schematic from the elaborated design produces a simpler diagram that shows grouped wires and merges each register into a single stacked group. Most of the logic gate images in this document were generated from the elaborated design. Figure 3 demonstrates the three types of components displayed in a typical elaborated schematic. Note that the elaborated schematic can be generated before synthesis occurs, but elaboration will fail if there are any syntax errors or missing modules.

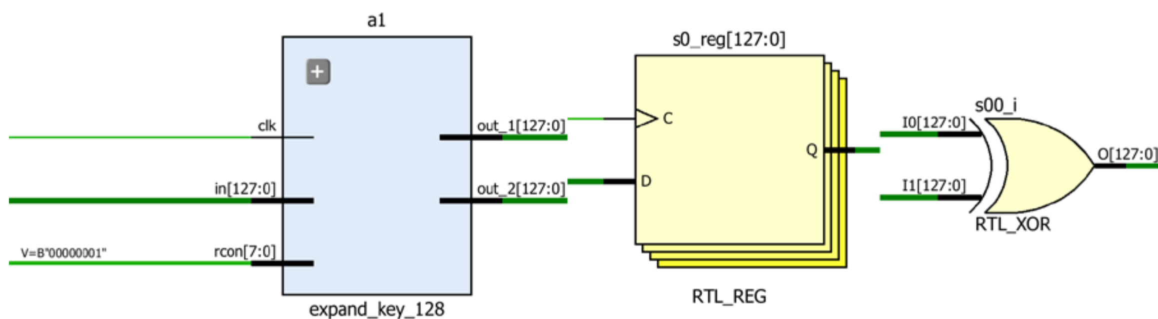


Figure 3. Components commonly displayed in a Vivado Schematic. At left is an example of a custom module. This module is defined as `expand_key_128` in the Verilog file `aes_128.v`. The specific instance is named `a1`. Note that clicking the + sign in the top left corner will produce an expanded view of this module, including all internal components. The middle item is a sample register, used to store values that are relevant to the circuit across multiple clock cycles. The stacking effect is a visual cue provided to represent a bundled multi-bit register. It is possible to unbundle this collection and produce a schematic with a separate register for each bit. The rightmost figure demonstrates Vivado's representation of primitive gates. Note that this XOR gate also handles multiple bits, but that no visual cue is given.

Once a schematic has been generated, it is possible to generate a simpler schematic based on selected wires and objects. Control-click allows you to select

multiple objects. Pressing `F4` will generate a new schematic containing the selected objects and any objects directly attached to them. Note that if you select a wire, the new schematic will include every gate, module and register connected to that wire. If the wire is used as an input for a large number of modules, the generated schematic will include all of those modules. For best results, build a new schematic from modules, registers and gates only. This will limit the number of additional features included in the schematic.

When a new, partial schematic is generated, you will notice that registers are split into individual, 1-bit flip-flops. This may cause the registers to be divided into 128 parts or more, resulting in a diagram that is not easily viewed on one screen. To correct this, open the waveform options pane using the top button on the left-hand toolbar. This pane includes an option labelled “Bundle Registers.” Uncheck this option, then check it again. The schematic will regenerate, with all registers combined correctly.

F. SIMULATION

The last stage of analysis was to trigger the circuits in behavioral simulation. Some circuits provided test benches designed to trigger the Trojan. These were simply added to the relevant projects as simulation sources and run. To add a simulation source, right click inside the sources window and select “Add Sources.” In the dialog box that appears, select “Add or Create Simulation Sources.” Selecting “Next” will open a dialogue that is similar to the “Add Sources” window from synthesis. Use “Add Files” to add a test bench to your project.

To run the test bench you have selected, click “Run Simulation” in the Flow Navigator. The default simulation settings are appropriate for the circuits discussed in this work, so you do not need to make any changes to “Simulation Settings.” After processing the test bench, Vivado will present a waveform diagram similar to that in Figure 4.

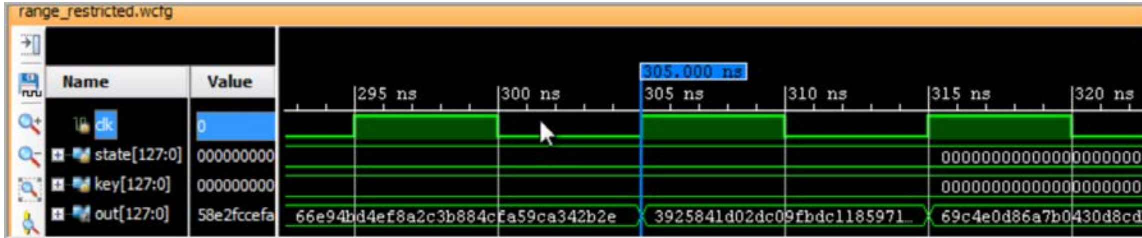


Figure 4. Part of the waveform diagram generated by Vivado from `test_aes_128.v` and the Trojan-free version of the AES circuit. Note that Vivado does not support changing the font size or the background color of the waveform area. Both features have been requested on the Xilinx support forums. Also note that the 1-bit input `clk` is represented by graphical lows and highs, but multi-bit inputs are represented by numerical values. For clarity, we have edited the colors of other waveforms in this document using external imaging software. This one has been left to demonstrate Vivado's default settings.

By default, the waveform diagram will display only the inputs and outputs from the unit under test. Additional circuit wires, such as `Tj_Trig`, can be added to the waveform window and saved in a new waveform configuration. To do this, use the schematic or the HDL code to determine the name of the wire you wish to inspect. In the "scopes" window, to the left of the waveform, select the module that contains the wire. The "Objects" window will list all nets in that module. Scroll down to the entry for the desired wire and drag it into the data region of the waveform window. This will add an entry to the window. Select "Save Waveform Configuration" from the left of the waveform window to save this change for future simulations of this project.

Note that when simulation first completes, the waveform diagram is zoomed in to the picosecond scale. The "zoom out" button at the left of the waveform window allows you to change the scaling to a larger timescale. The benchmarks in this set operate on a clock cycle of 1 to 10 nanoseconds, so zooming to the level shown in Figure 4 should produce the best results. Note that "save waveform configuration" will not retain your zoom setting. Every time you relaunch the simulation, you will need to adjust the zoom level.

The simulation will need to be restarted before the waveform diagram displays the value of new wires. Select “Run Simulation” again, and answer “Yes” when prompted to close the simulation and relaunch.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. BENCHMARKS

A. TROJANS IN AES_128

The Trojans in this set are based on an open-source implementation of a 128-bit AES encryption circuit. Each benchmark archive includes a folder `<archive top level>/src/TjFree`, containing 6 files. `Aes_128.v`, `round.v` and `table.v` contain the HDL code that defines the Trojan free circuit. These files are also present in each benchmark's `<archive top level>/src/TjIn` folder. The file `test_aes_128.v` is a test bench intended to demonstrate basic functionality of the circuit. This file is also present in most of the Trojan-inclusive variants. File `simulation.do` is a batch file designed for use with ModelSim. The last file is a `README` that provides some explanation for `simulation.do` and `test_aes_128.v`.

As Figure 5 demonstrates, the benchmarks in the AES series are modular in structure. In most of the benchmarks in this set, the `aes_128` module is an unmodified version of the original, Trojan-free AES circuit. Most AES benchmarks contain two additional modules. The first is named `TSC`, and is defined in the file `TSC.v`. This module contains the logic that defines the Trojan functionality. In several of the benchmarks, the `TSC` module and file have been renamed to `AM_Transmission` and `AM_Transmission.v`, respectively. Note that the `AM_Transmission` module, like the TSC module, operates in isolation from `aes_128` and accepts input `Tj_Trig` to determine part of its operation. The other module shown in Figure 5 is `Trojan_Trigger`, which controls when the functionality is active. This module is not always present in the benchmark. Its absence may represent an always-on trigger, but in some cases, Salmani et al. have instead written triggering logic into the `TSC` module.

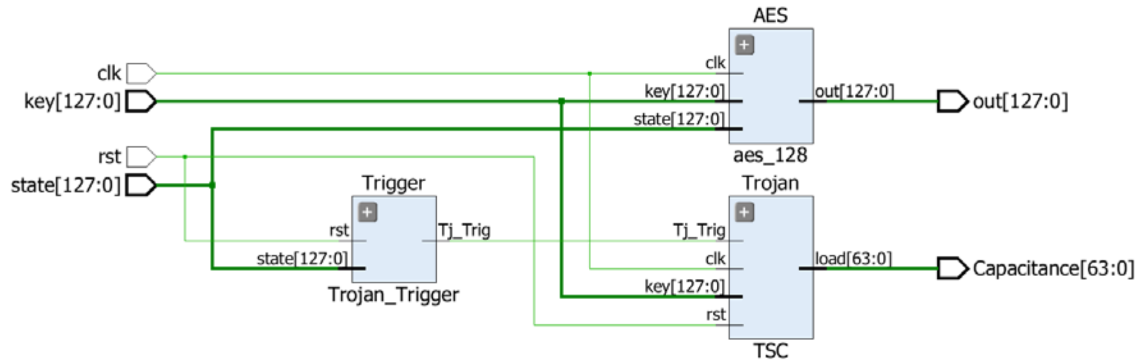


Figure 5. This schematic--generated from AES-T700--represents the structure of a typical AES Trojan from the collection. Note the modules for the trigger (Trojan_Trigger) and the Trojan functionality (TSC). These inclusions share common inputs with the Trojan-free aes_128 module, but they do not alter that module's internal operation.

1. Important Features of the Trojan-Free AES_128 Circuit

Figures 6, 7, and 8 demonstrate the workings of the Trojan-free AES circuit, including inputs, outputs, and the custom modules that run the encryption process. As shown in Figure 6, AES accepts three inputs, labeled clk, state, and key. Wire clk is drawn from the system clock. Bus state is a single 128-bit block from the plaintext message. Some of the Trojans in this collection use state as an input to the Trojan_Trigger module.

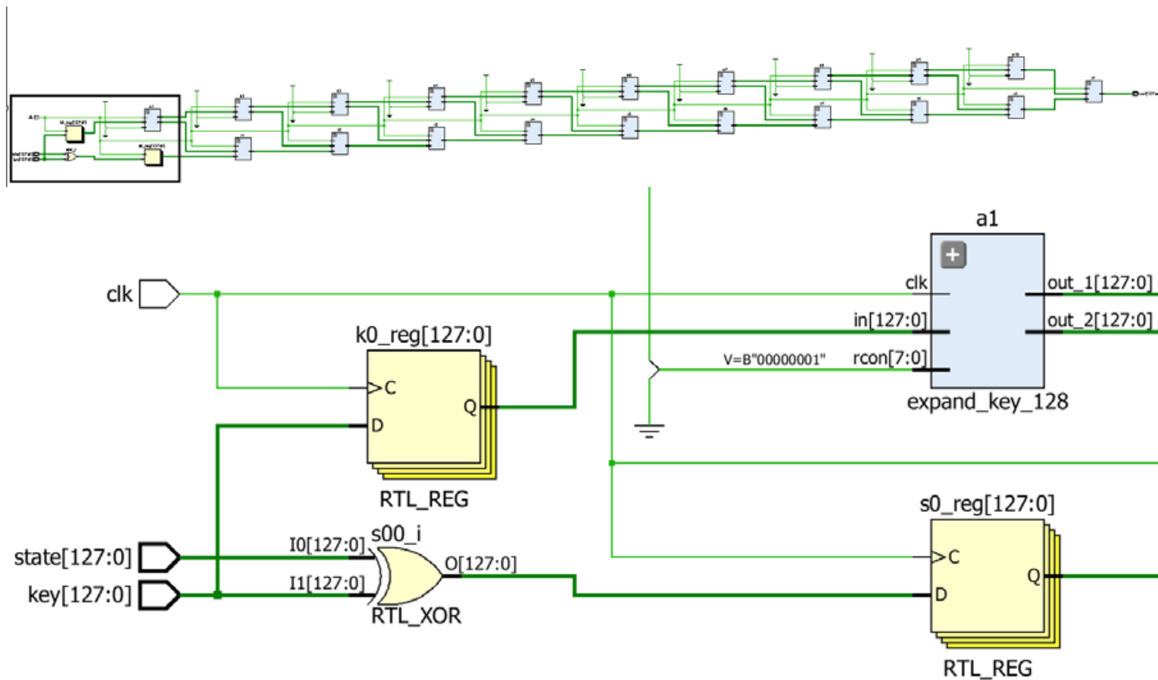


Figure 6. The input region of the Trojan-free AES implementation, complete with a world view of the circuit for context. The primary inputs to this circuit are `state` and `key`. Note that `state` and `key` are immediately XORed before the first round of the encryption process begins.

Bus `key` is a 128-bit symmetric key used for the encryption and decryption processes. The value of `key` is the most valuable information a Trojan can leak from the AES circuit. Viewing `state` reveals a single message to the adversary. Viewing `key` allows the adversary to decrypt all messages that have been sent using that particular key. Salmani et al. have designed most of the leakage Trojans in the AES collection to leak a portion of the key. Note that Figure 5 also shows an input labeled `rst`, which is applied to the `Trojan_Trigger` and `TSC` modules, but is not an input to `aes_128`. The reset signal `rst = ` 1_2 is used to revert the Trojan to its pre-activation state. In the test benches provided, `rst = ` 1_2 is transmitted for a several clock cycles before the first values of `key` and `state` are assigned. Wire `rst` is then set to 0_2 for the remainder of circuit operation.

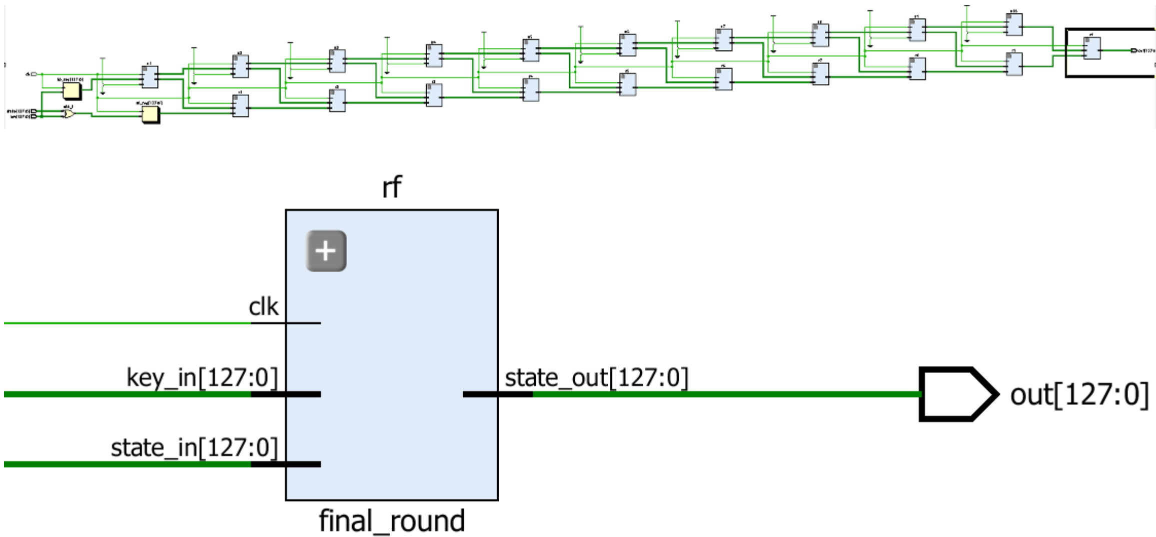


Figure 7. The output region of the Trojan-free AES circuit. The box labeled `rf` represents an instantiation of the custom module `final_round`, which is defined in the Verilog file `round.v`. This module represents the tenth round of the AES encryption process. This round is similar to previous rounds. It accepts the last round key and the last intermediate state and produces the circuit's final output.

Figure 7 displays the final output of the AES circuit, and the module that produces it. `Out` is the ciphertext resulting from the AES operation using the given key and state. Trojans in this group do not interfere with this value. Leakage in the AES benchmark circuits may be caused by the addition of a separate output bus, or by deliberately inducing electrical activity that can be observed by side-channel analysis. Denial-of-service in this circuit is caused indirectly, through the operation of a power-draining register. Exact details of these effects will be provided in the discussion of individual Trojans.

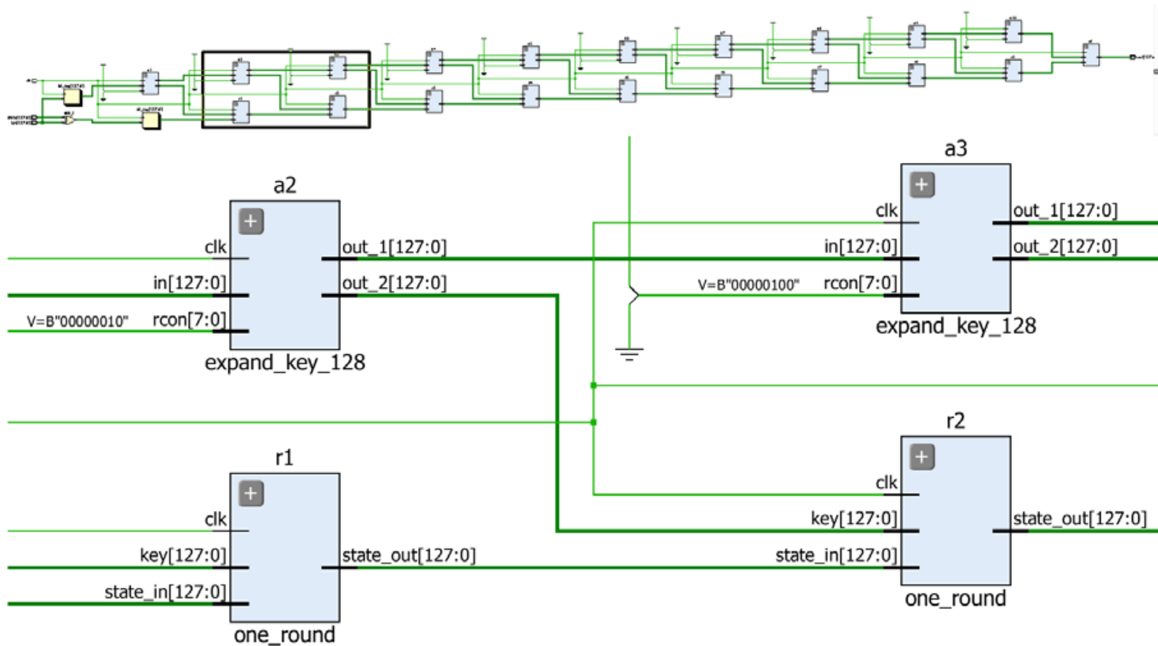


Figure 8. The internal functionality of the Trojan-free AES circuit. The `expand_key_128` modules generate intermediate round keys. Note that `out_1` and `out_2` hold the same value, so each round key is derived directly from the previous key.

In the AES process, `state` is transformed to `out` through ten rounds of substitution and translation. Each round uses a separate key, but each of these intermediate round keys is derived from the original AES key. Figure 8 shows two of these rounds. Each round produces the output `state_out`, which is used as `state_in` for the next round. Each round's `key` is generated by the `expand_key_128` module, using the key from the previous round. Several of the Trojans in this collection alter the `aes_128` module by adding additional module outputs, each of which leaks an intermediate round key to the `TSC` module. These intermediate keys are leaked through registers designed to be read by side-channel analysis.

2. AES-T100

The first few benchmarks in the AES collection use a slightly different structure than we discussed before. As Figure 9 shows, these benchmarks do not include a dedicated `Trojan_Trigger` module. Instead, the complete `TSC` module operates without any activation condition. These Trojans are classified as "always-on" Trojans.

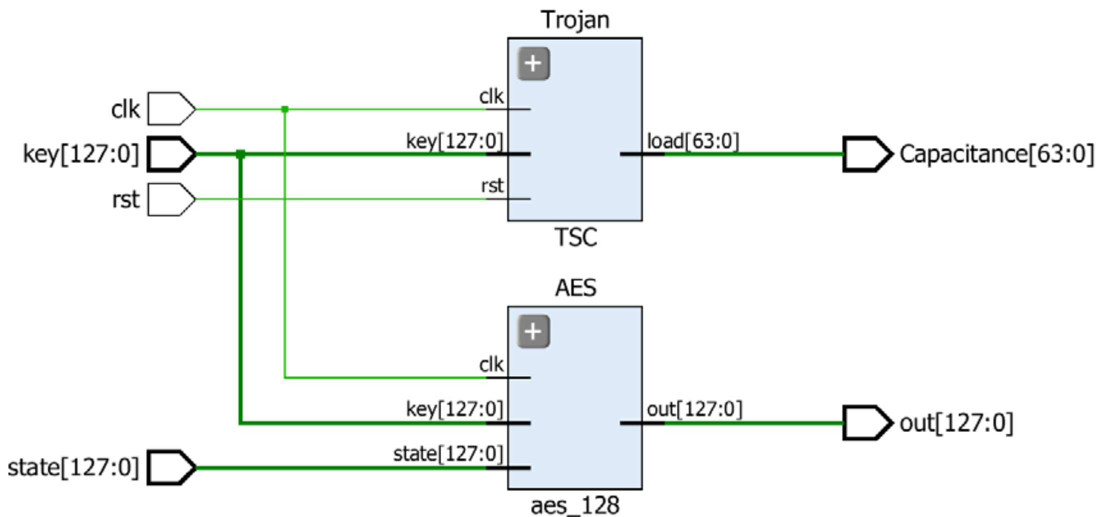


Figure 9. The structure of the benchmark `AES-T100`. Note the absence of the `Trojan_Trigger` module that was shown in the typical Trojan layout. No module is necessary to represent an always-on trigger.

a. Trigger

`AES-T100` does not have a trigger mechanism. As a result, all aspects of this Trojan, including the rotation of the LFSR, are always active.

b. Functionality

Figure 10 displays the internal functionality of the `TSC` module of `AES-T100`. The Trojan in this circuit leaks bits from the AES secret key. The adversary who created this Trojan intends to read the leaked bits using side channel analysis. To simulate sufficient capacitance for detection, each bit is actually leaked in parallel across 8 wires.

The 64-bit output load actually represents only 8 bits of leaked information. In fact, under this payload design, only the 8 low-order bits of key are ever leaked through load. Note that Figure 10 only explicitly shows the leakage of bits 0 and 7.

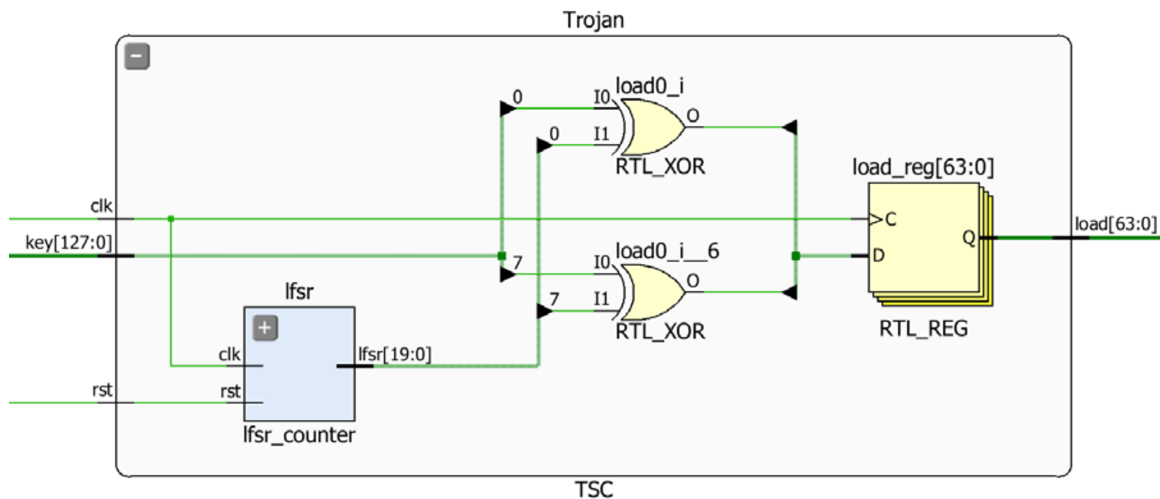


Figure 10. Detail view of the TSC module of AES-T700. The output load is composed of 8 bits, each repeated 8 times. These bits are the result of XOR operations between key[7:0] and lfsr[7:0]. To simplify this view, we have hidden the XOR gates load0_i__0 to load0_i__5. These XOR gates function in parallel to the two shown, with each gate operating on a different bit pairing.

Before being leaked, the bits of key are XORed with bits generated from the module lfsr. This modulation is designed to obfuscate the leakage, allowing only the adversary to translate side channel results into bits from the original key. Eight copies of each result are then fed to the register and transmitted to the capacitance circuit.

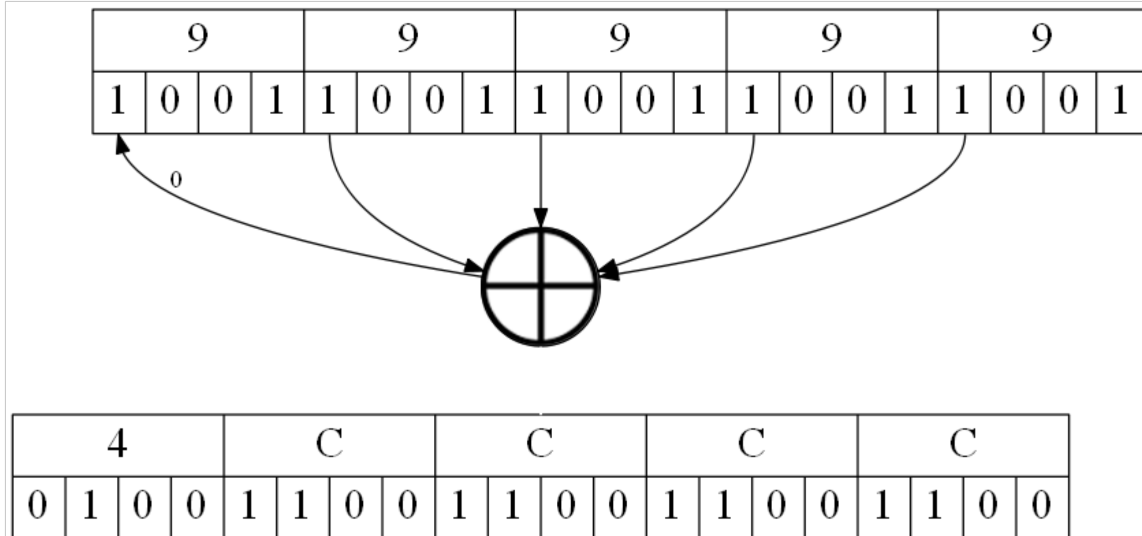


Figure 11. One shift operation of the `lfsr` module. The value `0x99999` is the initial value of the `AES-T700` LFSR register. The new high-order bit is calculated from an XOR of bits from positions 3, 7, 11, and 15. After this calculation, bits are shifted to the right, and the XOR result is used to fill in the missing bit.

LFSR registers are used in pseudo random number generation, because the sequence of values they produce appears to be random. However, an LFSR is actually guaranteed to enter a repeating cycle of values. If the adversary knows the initial vector of an LFSR, they will be able to predict future values. The LFSR in `AES-T100` is a 20-bit register with an initial value of `0x9999`. Figure 11 displays the first shift operation that will occur in this LFSR's operation. All bits shift to the right, and the missing bit is then filled based on a 4-way XOR operation. Note that the initial value is not part of the LFSR cycle. The first three values of this register are one-time events, as no value in the cycle will lead back to them. The value `0x13333` is the first value that can be considered part of the LFSR cycle. After 131,071 (2^{17} - 1) shifts, the cycle will restart at this same value.

3. AES-T200

a. Trigger

Like `AES-T100`, this benchmark doesn't have an explicit trigger mechanism. All of the Trojan functionality, including the rotation of the LFSR, functions at all times.

b. Functionality

As shown in Figure 12, the functionality of `AES-T200` is nearly identical to that of `AES-T100`. This module uses an LFSR to modulate the leaked bits of the key. This register is rotated every clock cycle without any need for activation. The distinction between this payload that in and `AES-T100` is the initial value of the LFSR. While `AES-T100` uses `0x99999` as an initial value, `AES-T200` takes its initial value from the incoming plaintext. When a reset signal is observed, the LFSR is loaded with the low-order bits of the input `state`. This feeding of an initial value allows the attacker to choose a different sequence to modulate the leaked bits of the key.

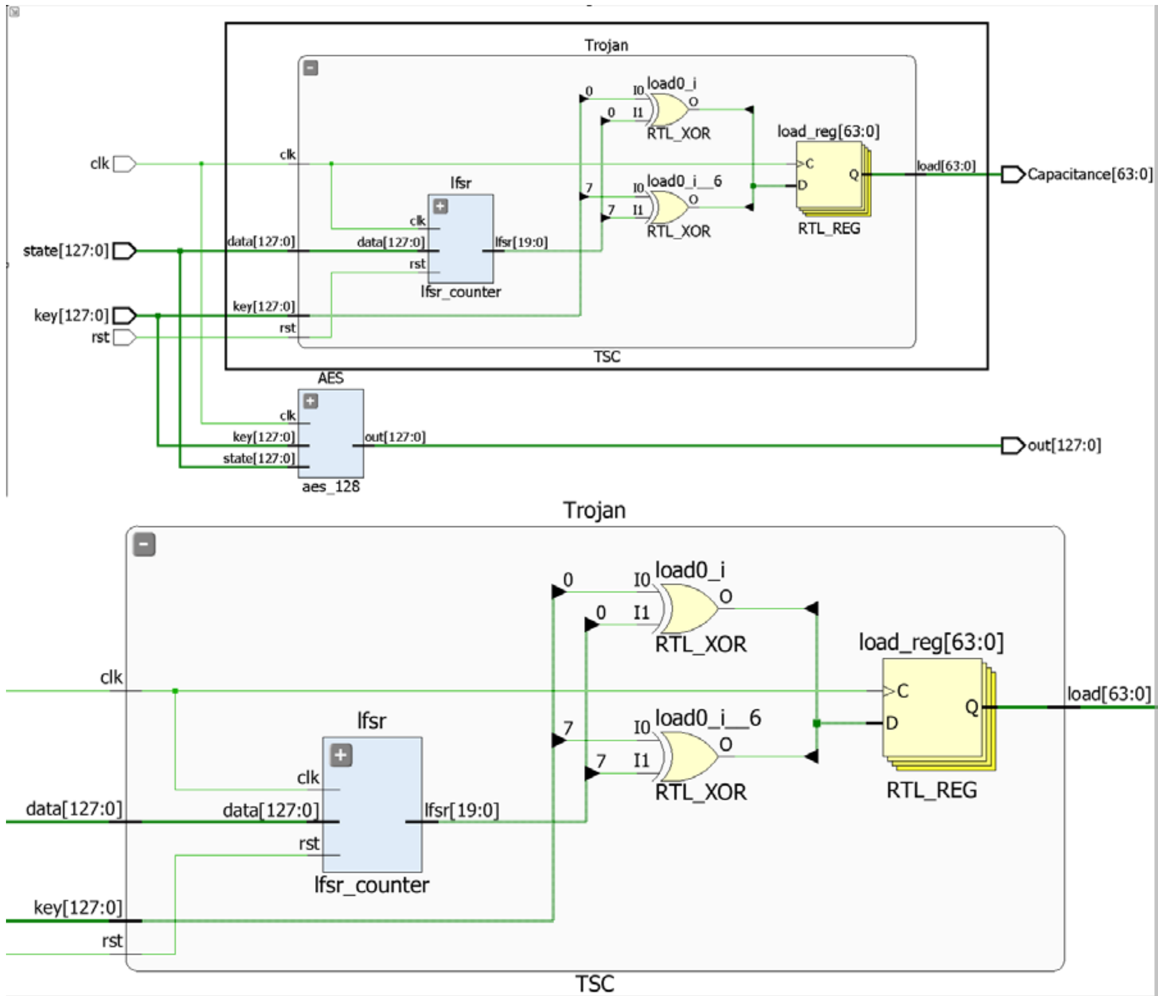


Figure 12. Detail view of the Trojan in `AES-T200`. Note the addition of the input `data` to the `lfsr_counter` module. The value of `data` is determined by `state`, one of the inputs to the overall circuit.

4. AES-T300

Most Trojans in the AES collection work in isolation from the main AES circuit. The trigger and functionality share common inputs with AES, but are otherwise completely isolated from it. The files that comprise the AES circuit are used in an unaltered state. In contrast, the `AES-T300` benchmark contains a modified version of module `aes_128`. This module has 8 additional outputs, each drawn from one of the intermediate round keys. Figure 13 shows a detail view of this. Note that the value of `rk8` comes directly from the `expand_key_128` module `a8`. The additional outputs are fed into the module `TSC`, which leaks them to the adversary.

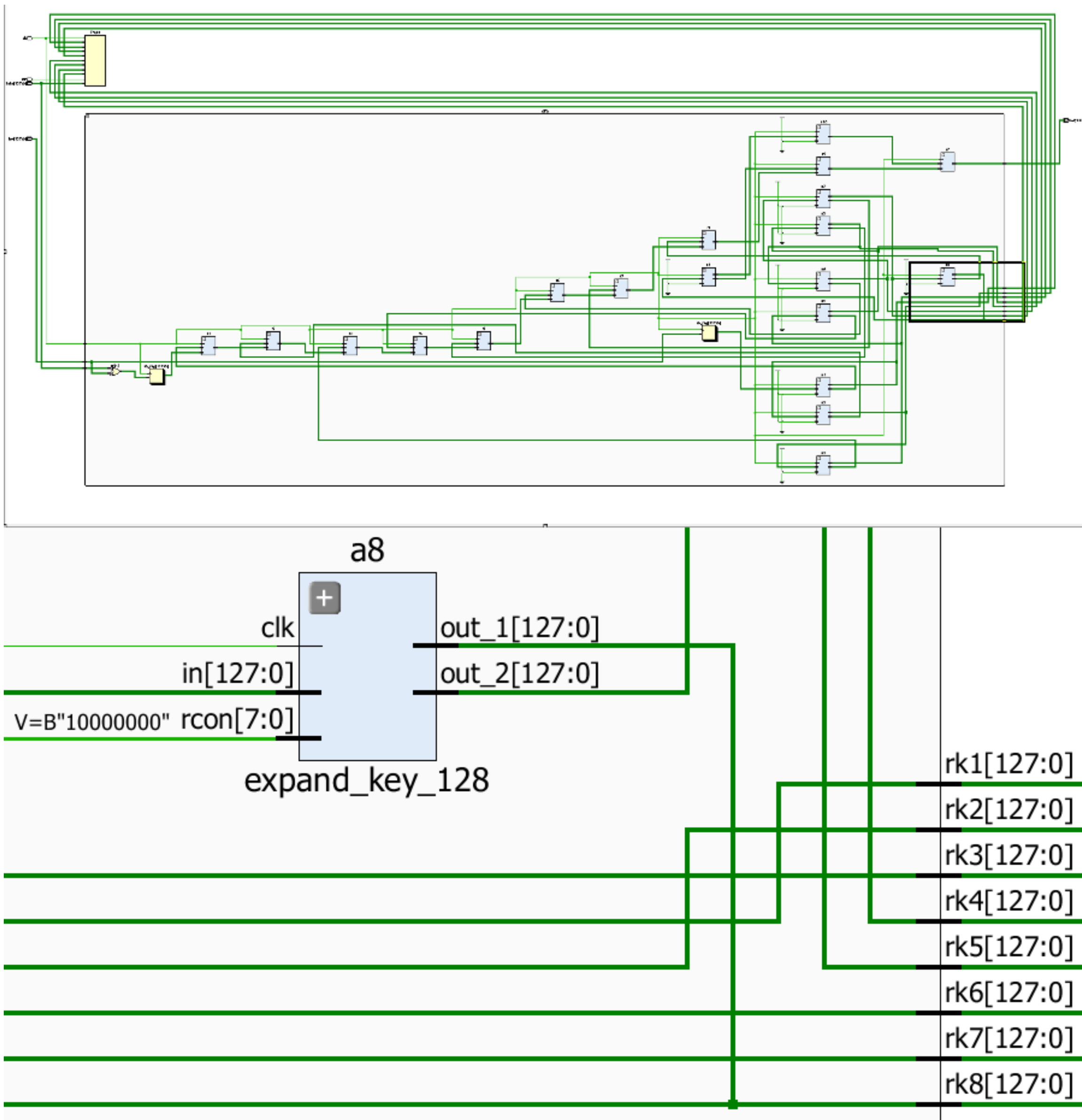


Figure 13. AES-T300 modifies the AES_128 module by adding eight additional outputs. These outputs carry eight of the round keys used during the AES encryption process. Note that the last two round keys are not leaked, but we assume that the attacker either knows or has the ability to discover the mechanism used to generate the key.

a. Trigger

This benchmark also uses an always-on trigger, meaning that the Trojan will be continuously active. The shift registers in this Trojan are dependent only on their respective round keys.

b. Functionality

The `TSC` module contains 8 8-bit shift registers like the one shown in Figure 14. Each register is dedicated to the leakage of a single round key. When a reset signal is observed, each register is loaded with the value $\texttt{10101010}_2$, or `0xAA`. The register rotates whenever `clk` changes and input `G` is equal to $\texttt{1}_2$. In `AES-T300`, this input is determined by performing a series of AND and XOR operations. The lowest 8 bits of state are ANDed with the lowest 8 bits of the round key. The resulting bits are subjected to an XOR operation. The final result of this operation is sent to the register. If this value is $\texttt{1}_2$, the register will rotate twice every clock cycle until the round key changes and the test is performed again.

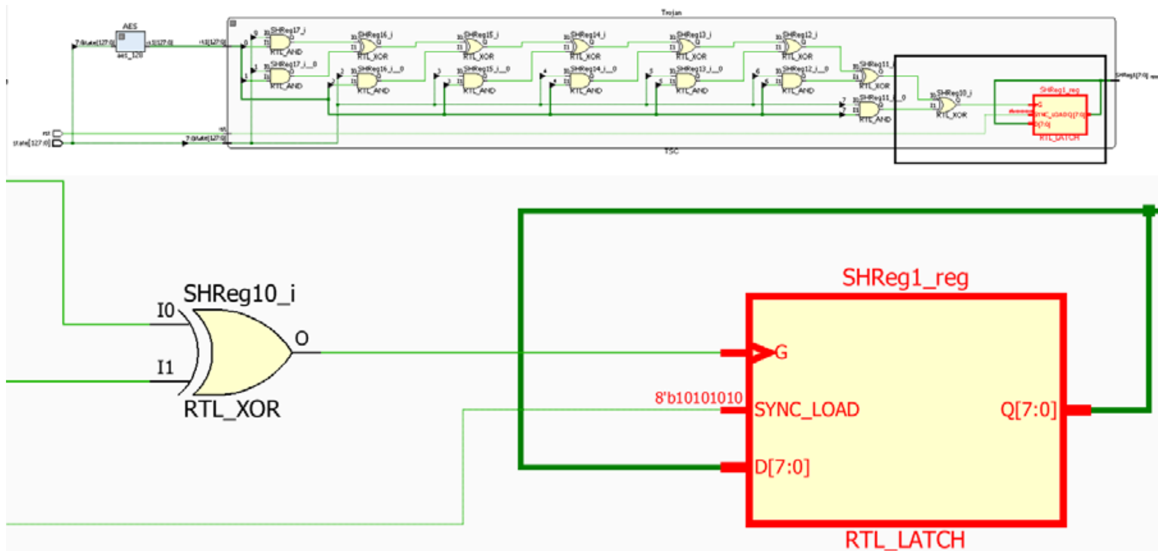


Figure 14. Detail view of a shift register from `AES-T300`. Note that the world view shown here is a partial schematic, depicting only 1 of the 8 registers. On a reset signal, the register is fed with an initial value of $\texttt{10101010}_2$. Afterwards, register's value remains unchanged unless the input `G` is $\texttt{1}_2$. `G` is the result of AND and XOR operations performed using state and the first round key.

Figure 15 shows a segment of the waveform for all eight registers. Note that the value of each register rotates between `0xAA` and `0x55`, which are the hex values of the

two alternating patterns for an 8-bit number. The adversary can identify the increased power consumption associated with the rotation.

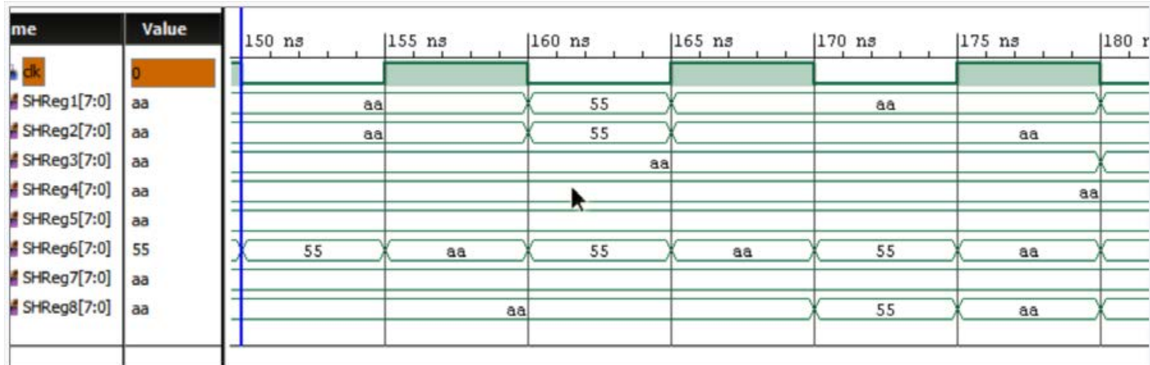


Figure 15. A waveform diagram displaying the actions of the shift registers in `AES-T300`. The top waveform displays the actions of the system clock. One step of rotation for a register means a transition from a value of `0xAA` to `0x55` or back. In the view shown, each round key has a different value, so the registers are rotating at different times. For example, `SHReg6` rotates every time the clock changes because it satisfies the AND and XOR test for the entire period shown.

Note that there is an error in the source code of the `TSC.v` file, which produces unexpected starting values in the registers.

```
if (rst == 1) begin
     SHReg1 <= "10101010";
```

The lines shown here cause the value `0x30` or 00110000_2 to be assigned to the first shift register. All 8 registers use this assignment statement, with consistent results. To correctly assign an alternating pattern to the register, change these lines to read:

```
if (rst == 1) begin
     SHReg1 <= 8'hAA;
```

A similar change will be required for each of the 8 registers.

5. AES-T400

AES-T400 is the first Trojan in this set to include a defined Trojan_Trigger module, as shown in Figure 16. This module evaluates the state of the circuit against the predetermined triggering condition, and if these match, then the wire Tj_Trig will transmit the signal 1_2 to the module TSC, signaling that the Trojan should begin to operate. Note that the signal rst=1_2 is typically used to reset the Trojan. When a reset signal is observed, the Trojan will be deactivated, and the trigger will wait for the next time the condition is met.

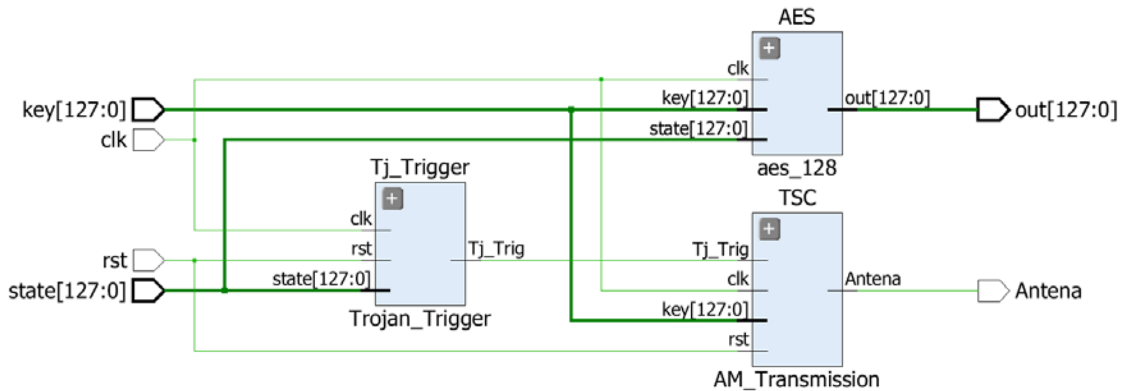


Figure 16. The full view of AES-T400. Note that this benchmark follows the typical structure displayed earlier. The functionality module is named AM_Transmission, but it accepts inputs from the wires key and Tj_trig. Like the TSC module present in other benchmarks, AM_Transmission operates without disrupting the core functionality of the aes_128 module.

a. Trigger

AES-T400 is described as a combinational trigger that activates when an input of 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF is observed. Figure 17 demonstrates this portion of the Trojan trigger mechanism.

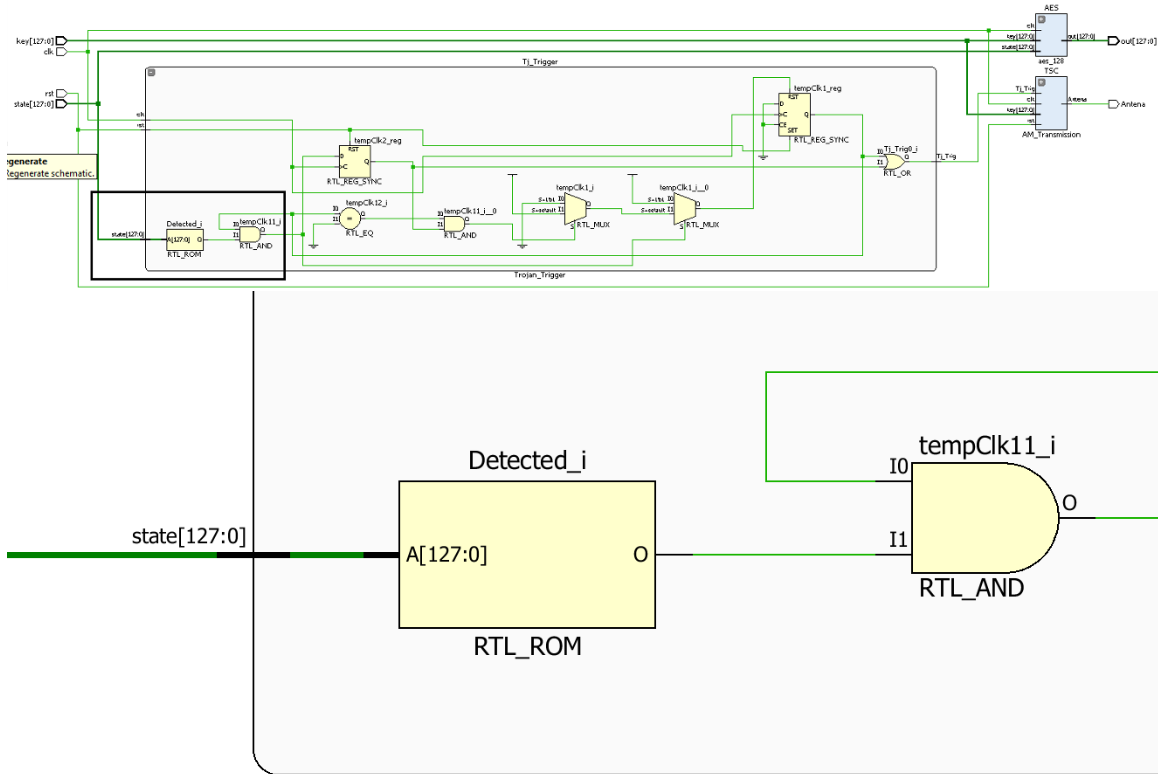


Figure 17. A detail view of the `AES-T400` trigger. `Detected_i` represents the actual comparison of the incoming value state against the predefined value of `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`. Note that due to features of `AES-T400`'s functionality, additional gates have been added to set `Tj_Trig` to 0_2 within two clock cycles of activation.

The activation of the `Tj_Trig` wire is designed to be temporary. While `Tj_Trig` has the value 1_2, the Trojan will load the value of input `key` into a register in the `AM_Transmission` module. However, the source code produces a different result. In the source code, the Trojan is activated when a reset signal is observed, and `Tj_Trig` becomes 0_2 one clock cycle later. It also becomes impossible to activate the Trojan after this point.

The following HDL code defines the activation of the AES-T400 trigger:

```
always @(tempClk1, tempClk2)
begin
     Tj_Trig <= tempClk1 | tempClk2;
end

// Tj_Trig is high for two clock cycles
always @(posedge clk)
begin
     if (rst == 1)  begin tempClk1 <= 1; tempClk2 <= 0; end
          else if ((tempClk1 == 1) && (Detected == 1))
               begin tempClk1 <= 0; tempClk2 <= 1;     end
          else if ((tempClk1 == 0) && (tempClk2 == 1))
               begin tempClk2 <= 0;     end
          else begin tempClk1 <= 0; tempClk2 <= 0; end
end
```

The first segment of code is used to determine the value on Tj_Trig. This value is decided by an OR operation between tempClk1 and tempClk2. If either wire is 1_2, the Tj_Trig will also be 1_2. Note that this assignment will occur every time one of the tempClk values changes.

The second code segment is responsible for assigning values to tempClk1 and tempClk2. This assignment will run at the start of every clock cycle. These two wires draw their values from four sources: their previous states, rst, and Detected. Detected is a 1-bit signal representing that state has been observed with a value of 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF since the last reset.

Note that when rst = 1_2, tempClk1 is also set to 1_2. This means that Tj_Trig is also immediately set to 1_2. On the next clock cycle, if state was not equal to 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF during the reset, tempClk1 and tempClk2 will be assigned a value of 0_2. This combination leads to a Tj_Trig value of 0_2 as well. Tj_Trig would hold a high value for only a single clock cycle. If the appropriate value of state is observed during a reset, then the high Trigger value will be maintained for two clock cycles. Note that the triggered portion of Trojan functionality is

the loading of a value into a register, and that event is scheduled to occur at the moment Tj_Trig is set to 1_2. Thus, the duration of the activation is not relevant.

After tempClk1 and tempClk2 are both equal to 0_2, a reset is the only incoming signal that can cause one of them to become 1_2. tempClk1 is only set to 1_2 by a reset, and tempClk2 is only ever set to 1_2 if tempClk1 and Detected are simultaneously equal to 1_2.

b. Functionality

AES-T400 uses a new leakage mechanism. The LFSR leakage circuits in AES-T100 and AES-T200 rely on side channel analysis. The Trojan in this benchmark actually succeeds in transmitting information to an AM radio at 1560 kHz. If you choose to implement this circuit on physical hardware, note that the documentation does explain how to interpret the signals received on this radio frequency. Salmani et al. state that a single beep followed by a pause represents a 0_2, and a double beep represents a 1_2. Operating only within Vivado, we have been able to observe the operation of the output Antena [sic], which is the line that carries signals to the transmitter itself. Based on the structure of the HDL code, a 1_2 value of Antena [sic] results in a beep, and a 0_2 value results in silence.

When the Trojan is first activated, the value of the secret key is loaded into a register named ShiftReg. The structure of this register is shown in Figure 18. When the Tj_Trig wire is 0_2, this register will shift once for every full rotation of the 26-bit counter Baud8GeneratorACC. This register increments every clock cycle. As a result, ShiftReg will rotate one step every 2^{128x26} clock cycles. The least significant bit of this ShiftReg will be leaked to the transmitter, resulting in a transmission of the key in reverse order.

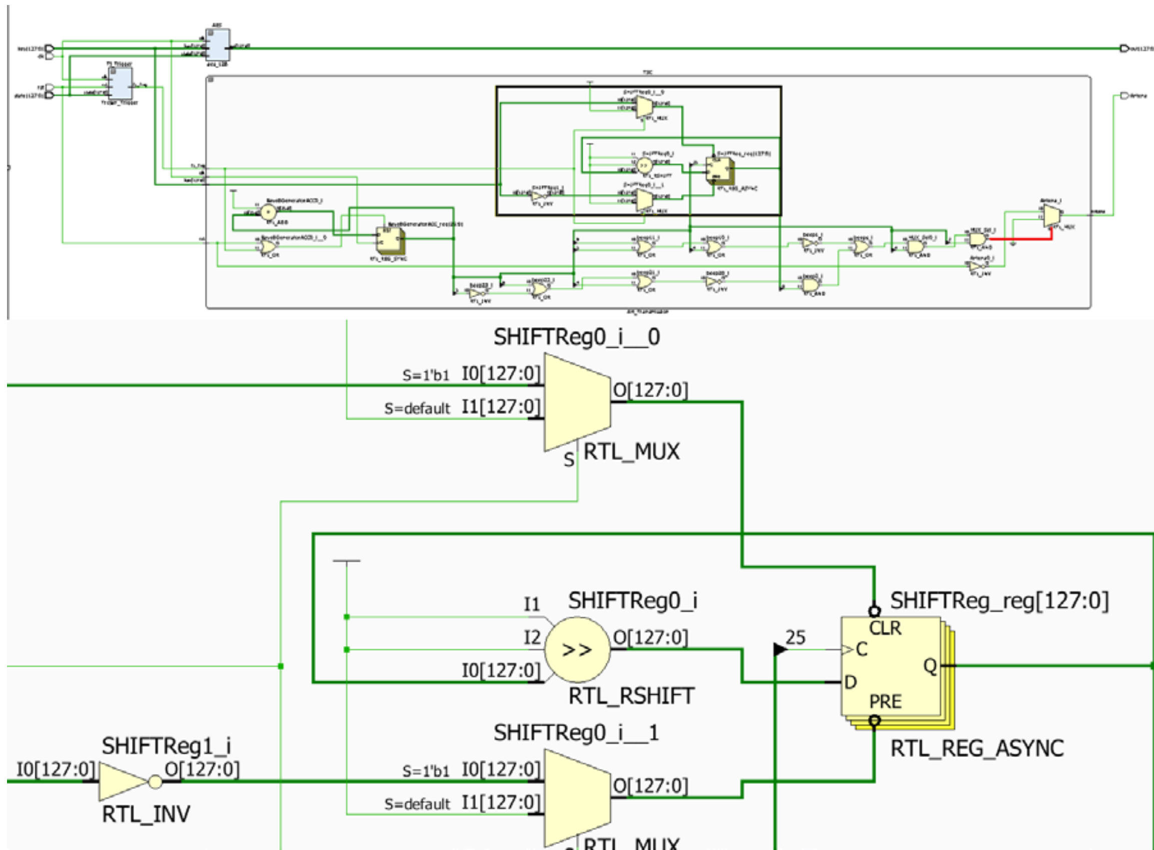


Figure 18. The shift register of AES-T400. When Tj_Trig = 1_2, this register is loaded with key. Every time that the register's C input is set to 1_2, the register input is updated with the output from SHIFTReg0_i. This output is the register's previous value with every bit shifted to the right.

The beeps are partly controlled by Baud8GeneratorACC. A first beep requires that the register contain a value of the form 0x000x…x1x…x1xxxx. The 1_2 signals are bits 4 and 15 of the register. The second beep occurs when Baud8GeneratorACC has the form 0x010x…x1x…x1xxxx and the ShiftReg[0] bit is 1_2. The change in the value of bit 24 results in a short period of silence in a double beep, while the requirement for a 0_2 value in bit 25 means that after the period allowed for the second beep, Baud8GeneratorACC must wrap around before the next bit is transmitted. This makes the pause between bits much longer than the pause in the middle of a pair.

6. AES-T500

At first glance, AES-T500 does not appear to have a trigger module. The benchmark does not contain a Trojan_Trigger.v file, and, as Figure 19 shows, the benchmark circuit doesn't explicitly define a module for the trigger. However, TSC.v contains a section of code that is an exact match for code in one of the Trojan_Trigger modules in AES-T800. The code even uses the name Tj_Trig to define the signal to the module payload.

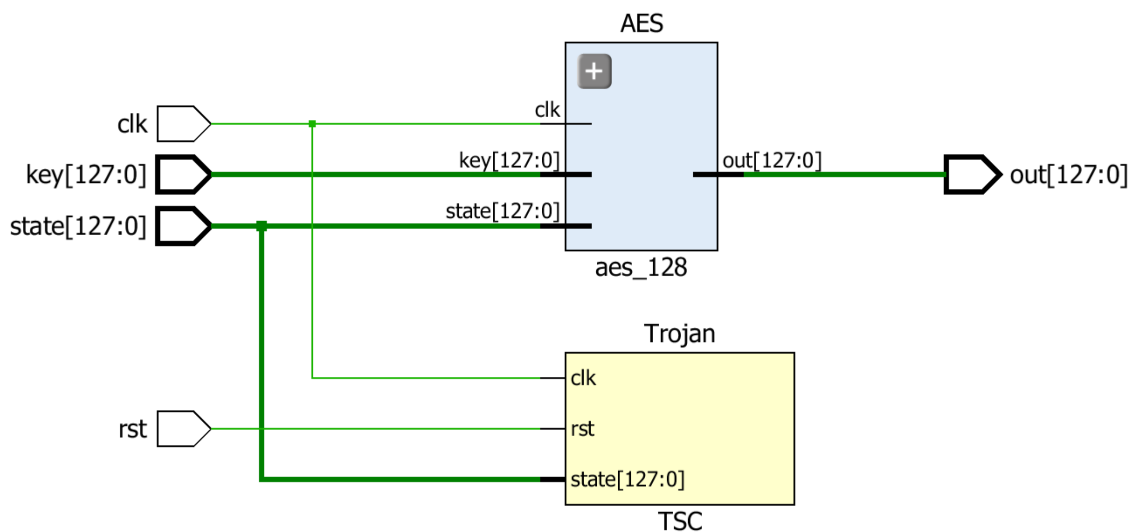


Figure 19. The apparent structure of AES-T500. Note that while no trigger module has been explicitly defined, the TSC module contains logic dedicated to the purpose of triggering the Trojan after specific inputs have been observed. Also note that Vivado has not elaborated on the contents of the module TSC. Vivado takes this approach to modules that do not have a specific output. To work around this, we added an output to the TSC module, directly using an existing register to provide the output's value. With this change, we were able to generate detail views of the internal workings of the TSC module.

a. Trigger

The trigger in this benchmark is a sequential trigger. This trigger waits for four separate values of state to be observed, in order, before setting `Tj_Trig` to 1_2. The inputs are as follows:

- `0x3243f6a8_885a308d_313198a2_e0370734`
- `0x00112233_44556677_8899aabb_ccddeeff`
- `0x0`
- `0x1`

Note that these inputs do not need to be observed in immediate succession. After each value is observed, a state register is updated. The value of this register must be 1_2 for the next input value to move the sequence forward. Hundreds of state values could be observed between the first of the triggering values and the last. As long as the circuit is not reset, the Trigger mechanism will remember which of these states have been seen. Figure 20 shows the first two state comparisons. The register `State0_reg` identifies whether the first triggering input combination has been seen since the last reset. Note that the result of the `state11_i` comparison is ANDed with the register value. If they are not both 1_2, then the next state register will not be updated.

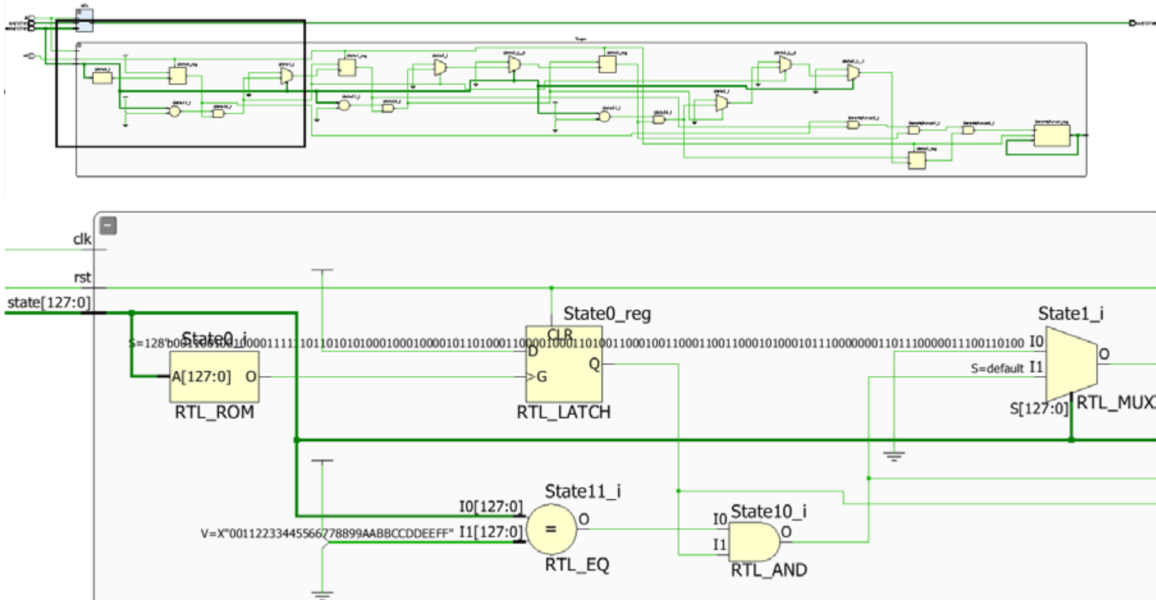


Figure 20. A detail view of the first two comparisons performed by the trigger in AES-T500. Vivado represents the first comparison as the RTL_ROM module State0_i. This module is slightly overshadowed by the long binary input to State1_i. Note the result of this comparison is stored in the RTL_LATCH State0_reg. The second comparison is depicted as a direct comparison of state to a given value using an RTL_EQ module.

b. Functionality

AES-T500 is designed to perform a denial-of-service attack. Recall that AES-T300 uses rotating registers to leak information to an adversary. This leakage is possible because the rotation of an alternating register increases power consumption. Side channel analysis allows the adversary to recognize that consumption and deduce information about the key. AES-T500 uses register rotation, but there is no intent to leak information. Instead, this Trojan is meant to be used on a battery-operated device. With the register rotating every time the clock changes, the device experiences an increased drain on the battery. As a result, the device will need more frequent recharges. Note that Figure 21 shows the full functionality of the device. The majority of the gates and registers shown in the world view are part of the benchmark's trigger mechanism. Only DynamicPower_reg can be considered part of the functionality of this Trojan.

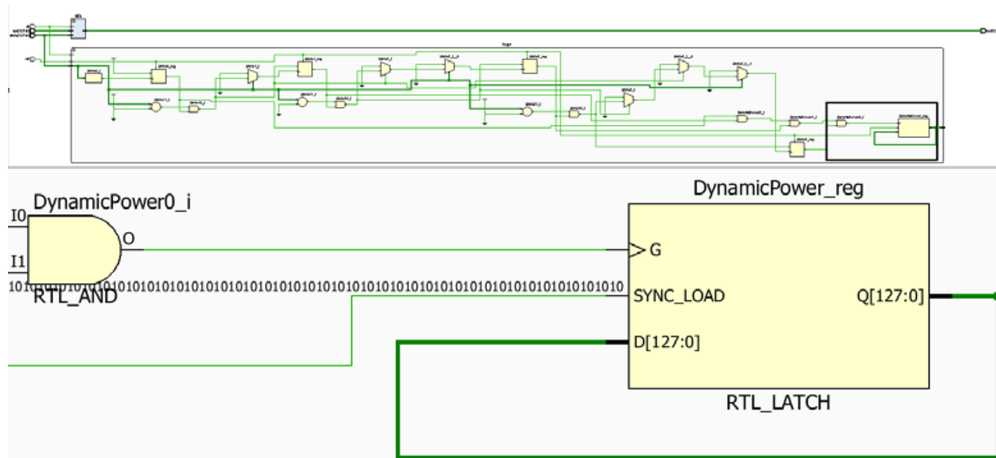


Figure 21. Detail view of the Trojan functionality of AES-T500. DynamicPower_reg is initialized with an alternating pattern of 1010…$_2$. The wire feeding into G is Tj_trig. When this wire has a value of 1_2, the register will rotate on every clock cycle. The bus from Q to D represents the altered value being fed back into the register. Note that this view cuts off the extra output we added to force Vivado to generate this image.

7. AES-T600

a. Trigger

AES-T600 uses the same trigger as AES-T400. The Trojan_Trigger.v files differ only in the placement of whitespace. This trigger has a mechanism for detecting a state value of 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF. However, the trigger activation actually occurs at reset. Note that if the expected state value occurs at the same time as the reset signal, then the Tj_Trig wire will remain 1_2 for two clock cycles. Otherwise, the trigger will return to 0_2 after 1 clock cycle. It will not be possible to activate the trigger again without another reset signal.

b. Functionality

The Trojan in AES-T600 leaks bits from the secret key. Unlike many of the other AES Trojans, this Trojan will actually leak the entire key, one bit at a time.

When the wire Tj_Trig first shows a 1_2 value, the key from AES will be loaded into a register named SECRETkey. Since Tj_Trig quickly changes back to 0_2 and the

Trojan cannot be triggered again until a new reset, SECRETkey will retain the secret key assigned at reset for the remainder of the leakage process.

When the value of SECRETkey changes, the least significant bit of the new value is fed to an inverter. The result is inverted again, and split across ten wires, as shown in Figure 22. These ten wires are meant to be read using side channel analysis. The result of this inversion is meant to be read through side channel analysis. Like the LFSR-modulated leakage used in AES-T100 and AES-T200, the analysis is easier when the bit is repeated across multiple wires.

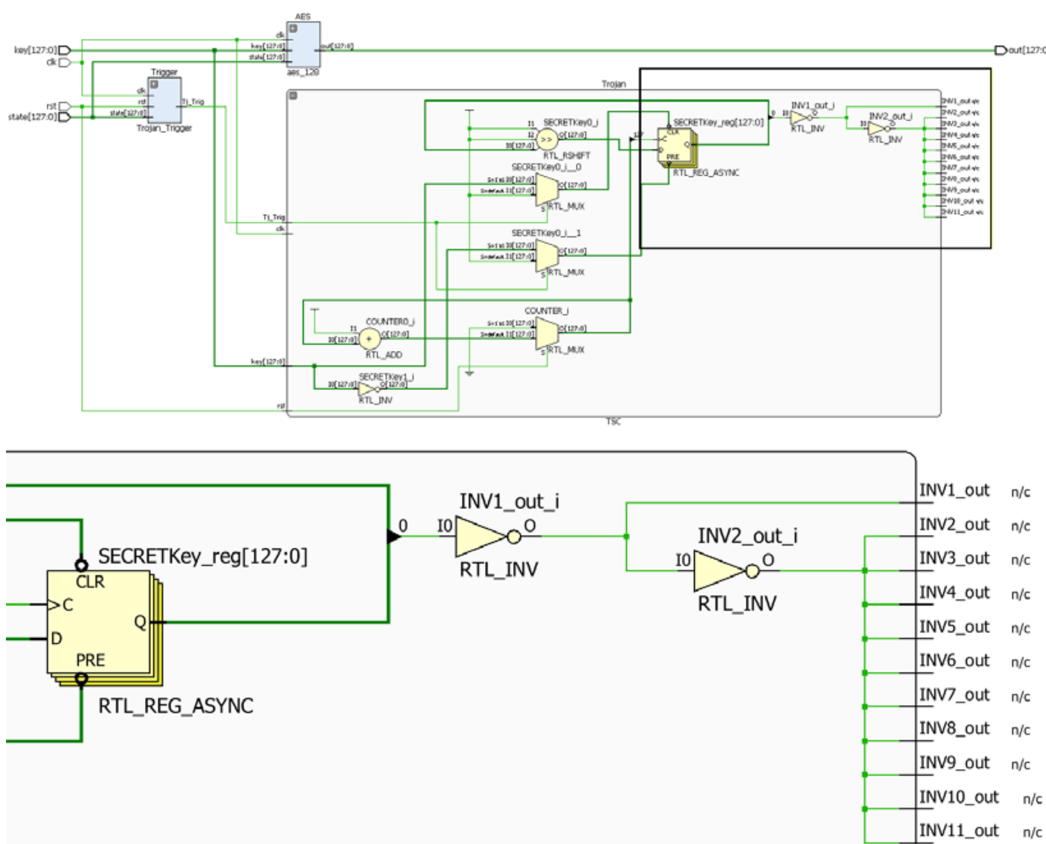


Figure 22. Leakage from the AES-T600 Trojan. Note that the wires INV1_out through INV11_out are defined as wires, but not circuit outputs in the Trust-Hub code. According to the documentation for this benchmark, the leaked bit can be recovered by measuring the leakage current. To produce this schematic, we extended the wires into module outputs. Without the change, Vivado would refuse to expand the TSC module.

To leak the rest of the secret key, AES-T600 uses SECRETkey as a shift register. A different 128-bit register is set to 0_2 when a reset signal is observed. Every clock cycle, this register increments. When the most significant bit becomes 1_2, the bits of SECRETkey shift one place. Since SECRETkey has just changed, the inverters now leak the new least significant bit. In this way, the key is leaked, in reverse order, to the side channel the adversary has prepared.

8. AES-T700

a. Trigger

The trigger of AES-T700 operates by a single comparison operation. Figure 23 demonstrates the Vivado schematic of this comparison mechanism. Each value of state is compared against the value 0x00112233445566778899AABBCCDDEEFF. Note that Vivado represents this comparison as a ROM access with only one of the 2^{128} memory locations containing a value of 1_2.

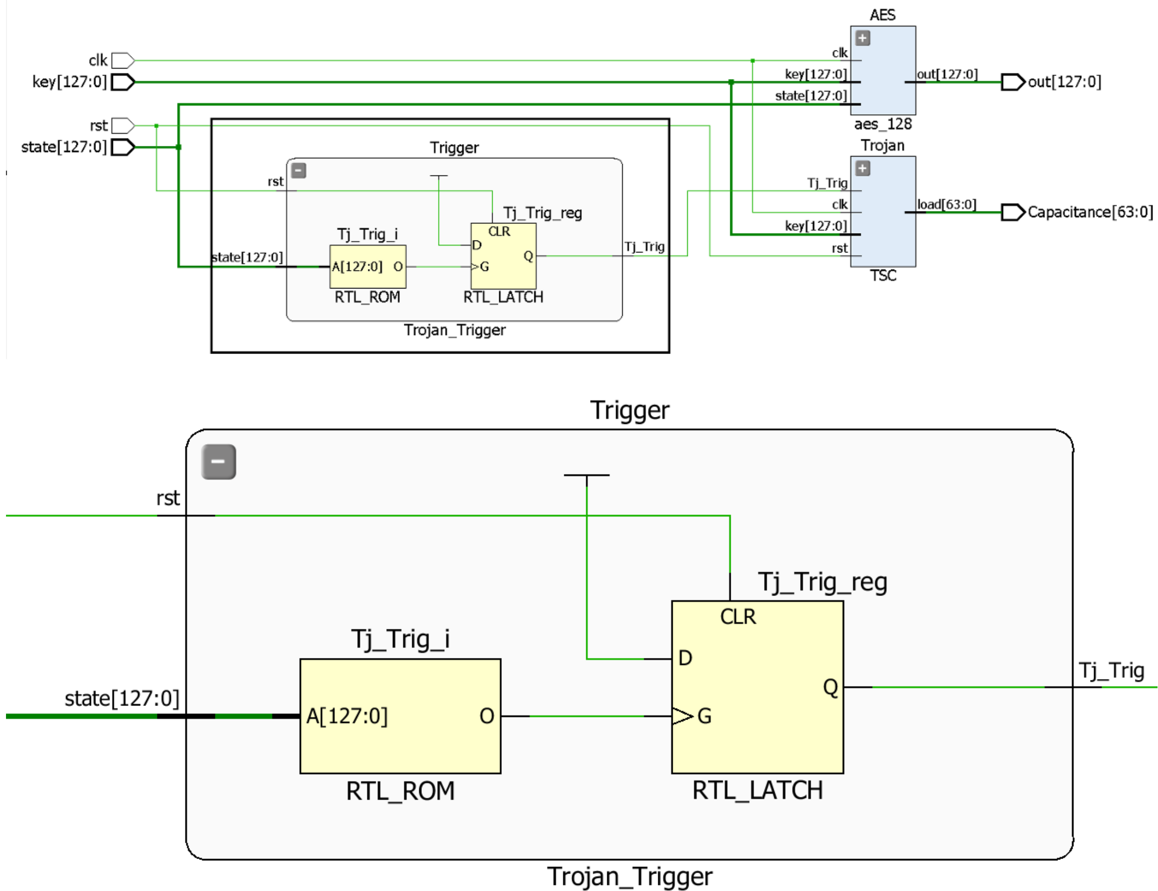


Figure 23. Detail view of the `Trojan_Trigger` module of `AES-T700`. Activation is represented by a 1_2 signal on the output `Tj_Trig`. This signal begins when the correct input state is observed, as compared to the value in `Tj_Trig_i`. A 1_2 signal on `rst` will deactivate the Trojan by acting as the "clear" input of the `RTL_LATCH`.

When a state with the chosen value is observed, the wire `Tj_Trig` will be set to 1_2. This signal causes the Trojan module to activate. The Trojan will remain active until a signal of 1_2 is observed on the `rst` input, at which point the `Tj_Trig` will be set back to 0_2. Vivado represents the activation process using an S-R latch, identified as an `RTL_LATCH`. The output of the ROM comparison serves as the set signal for the latch. The value of `Tj_trig` is determined at the instant that one of the incoming signals changes, not at the start of a clock-cycle.

b. Functionality

AES-T700 uses a leakage circuit nearly identical to that used in AES-T100. The LFSR has identical functionality, and the same initial value of 0x99999 is loaded at the rst signal. The only difference between the TSC module here and the TSC module in AES-T100 is the addition of the input Tj_Trig, which serves to signal the activation of the Trojan.

In Figure 24, you can see that the wire Tj_Trig is only applied to the module lfsr. The XOR and leakage operations will occur regardless of whether or not the Trojan has been triggered. This means that the leaked bits will be repeatedly XORed with bits from the initial value of the LFSR—0x99999. Trojan activation in this circuit will cause the LFSR to begin operating, improving the leakage obfuscation.

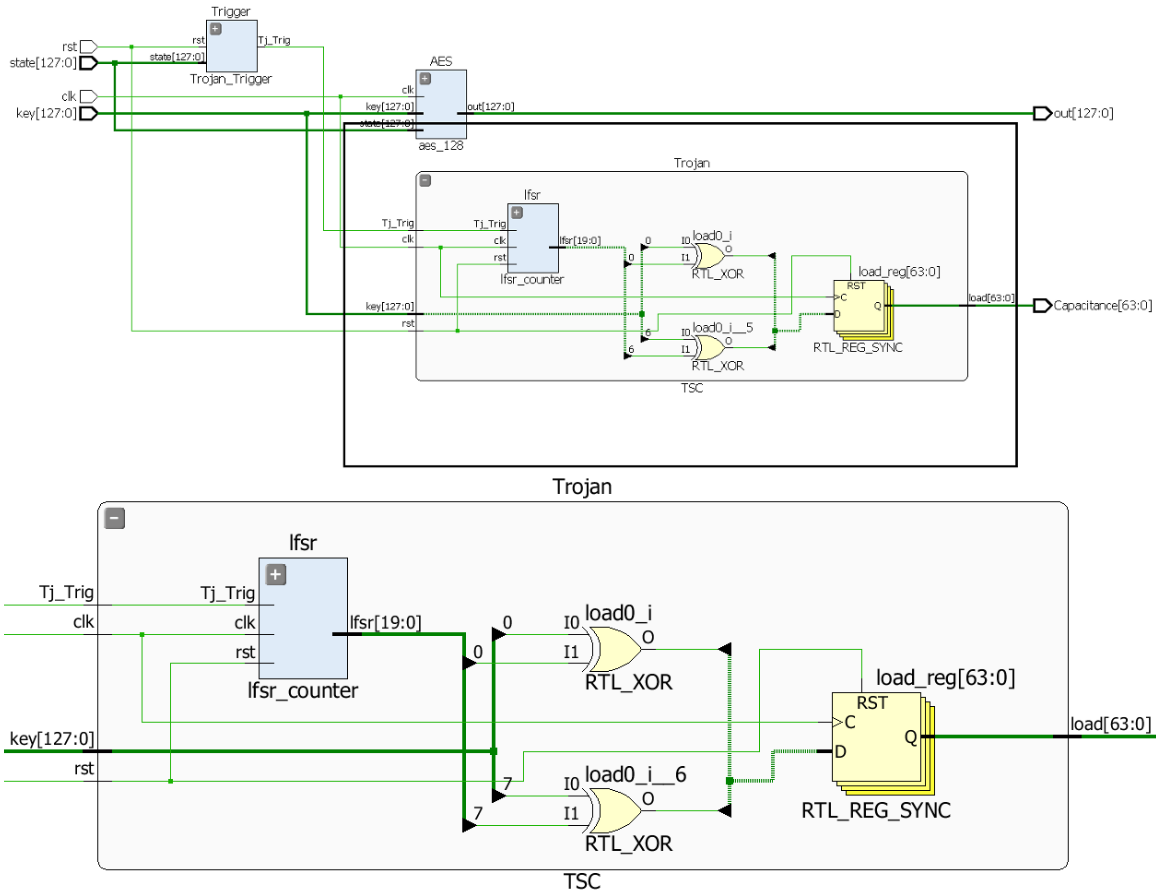


Figure 24. Detail view of the TSC module of AES-T700. Note the presence of the Tj_Trig input. This input only applies to lfsr, meaning that only the operation of that module is affected by the activation of the Trojan.

9. AES-T800

The AES series of benchmarks was designed in a very modular fashion. All of the benchmarks from this point forward directly copy their trigger, functionality or both from a previous circuit. As a result, we have included only a quick summary of duplicate elements, along with a reference to the benchmarks they have been borrowed from.

a. Trigger

The `AES-T800` circuit uses a sequential trigger identical to that in `AES-T500`. The Trojan will be activated after the trigger module observes the sequence `0x3243f6a8_885a308d_313198a2_e0370734`, `0x00112233_44556677_8899aabb_ccddeeff`, `0x0`, `0x1`. However, `AES-T800` actually dedicates a module completely to this mechanism, instead of including the trigger as part of the TSC module.

b. Functionality

`AES-T800` uses the exact same `TSC` module as `AES-T700`. This can be confirmed through hash comparison of `TSC.v`, which contains the logic for this module. The MD5 hash of this `TSC.v` is `0x9EA2ADE64F51E044AF4056C6B0F487E0`. The module leaks 8 bits of the secret key after modulating them with bits generated by an LFSR. The LFSR is loaded with an initial value of `0x99999`. Note that the XOR operation and the leakage will occur every clock cycle, but the LFSR will not rotate unless the Trojan has been triggered.

10. AES-T900

a. Trigger

`AES-T900` uses a 128-bit counter to determine the status of `Tj_Trig`. The functionality of this counter is shown in Figure 25. The counter is assigned a value of 0_2 at reset. Every time an encryption is completed, the counter is incremented. When the counter has a value of `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`, the circuit is triggered.

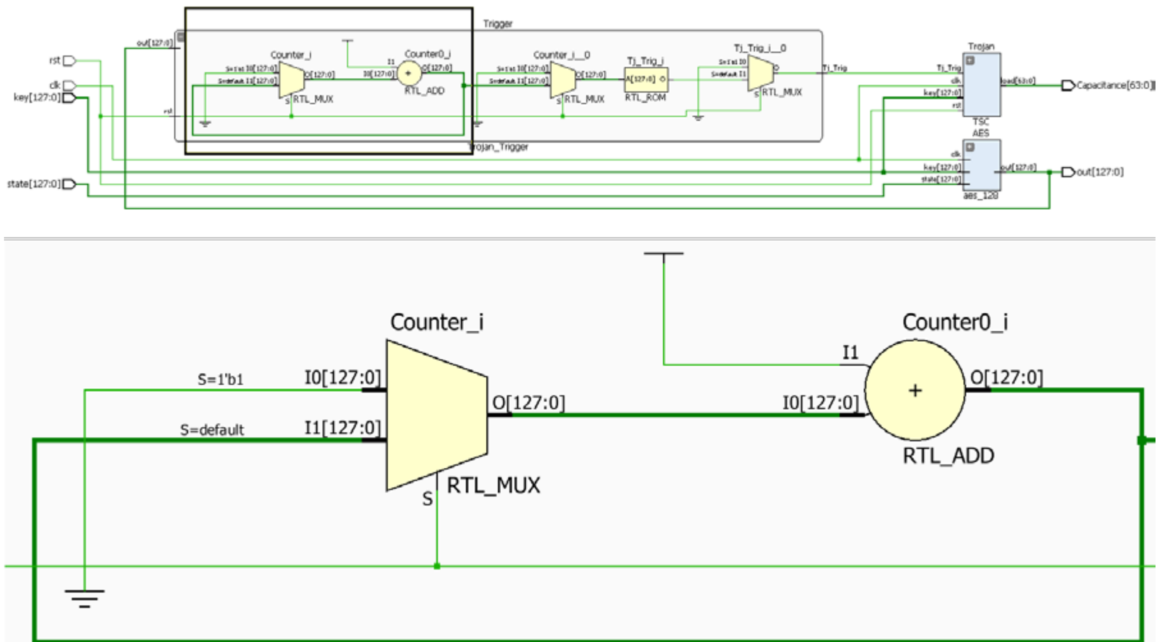


Figure 25. The counter in `AES-T900`. Vivado is not representing this counter using a register module. Instead, the value of the counter is represented by the loop passing between `Counter0_i` and `Counter_i`. Note that the select bit of `Counter_i` is provided by `rst`.

b. Functionality

`AES-T900` uses the same `TSC` module as `AES-T700` and `AES-T800`. An LFSR with an initial value of `0x99999` is used to modulate bits from the secret AES key. The bits are then leaked to a capacitance circuit. The adversary can read the leakage using side channel analysis. Since the adversary determined the initial vector and the LFSR mechanism, they will be able to convert the bits back to their correct values. Note that this Trojan only leaks the least significant 8 bits of the key.

11. AES-T1000

a. Trigger

The trigger mechanism is identical to that uses in `AES-T700`. Both benchmarks use an identical version of `Trojan_Trigger.v`, with an MD5 hash of `DF55995DD60E427AFB27050A0B6D21DD`.

After the value 0x00112233445566778899AABBCCDDEEFF is observed, the wire Tj_Trig will be set to 1_2. A reset signal will return the value to 0_2.

b. Functionality

The functionality of AES-T1000 mixes the functionality of AES-T200 and AES-T700. The LFSR register is rotated every clock cycle, but only after activation, as represented by Tj_Trig = 1_2. The initial value of the LFSR is taken from the input plaintext, just as in AES-T200. While AES-T700 uses 0x99999 as an initial value, AES-T1000 takes its initial value from the incoming plaintext. This value is loaded at reset.

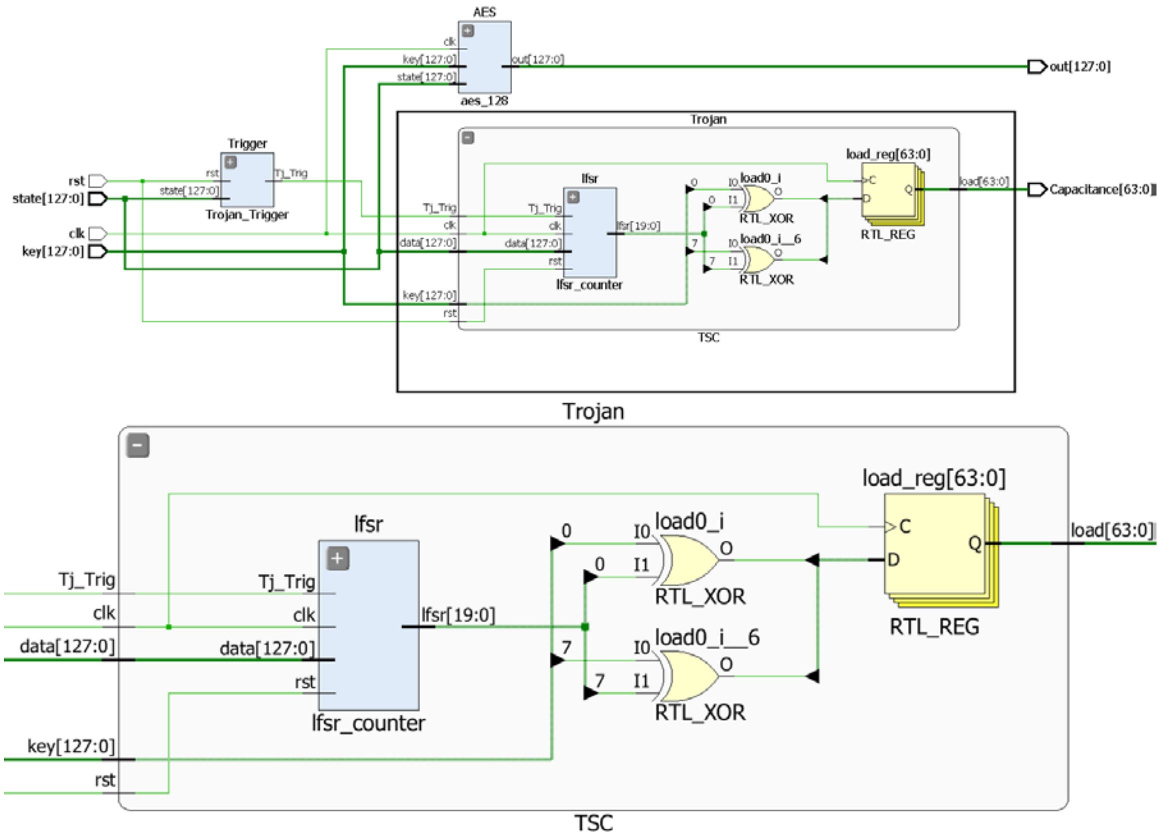


Figure 26. Detail view of the TSC module in AES-T1000. Note that this module combines features from AES-T200 and AES-T700. The LFSR register is initially fed from data, which is state in the overall circuit. Tj_Trig is used to control only the rotation of the LFSR.

12. AES-T1100

a. *Trigger*

AES-T1100 uses the same trigger as AES-T500 and AES-T800. Note that the Trojan_Trigger.v files of AES-T800 and AES-T1100 share an MD5 hash of 403360F99B64A4524058DEC3268AE5AC. This trigger activates after observing the following sequence of state values in order, but not necessarily in immediate succession:

- 0x3243f6a8_885a308d_313198a2_e0370734
- 0x00112233_44556677_8899aabb_ccddeeff
- 0x0
- 0x1

b. *Functionality*

AES-T1100 copies the functionality of AES-T1000. These benchmarks share a common TSC.v file with an MD5 hash of 0x3A8620C109A5D632A0448C6DFE996D2C. Note that this module leaks bits from the secret key after XORing them with bits from an LFSR register. This variant of the LFSR register is more flexible because it draws its initial value from state. This value is assigned when a reset signal is observed.

13. AES-T1200

a. *Trigger*

AES-T1200 uses the same counter-based trigger as AES-T900. A 128-bit counter is assigned an initial value of 0x0. Every completed encryption causes the counter to increment. When the counter's value is equal to 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF, the wire Tj_Trig will begin carrying the 1_2 signal that represents activation.

b. *Functionality*

This TSC module is defined in the same version of TSC.v used in AES-T1000 and AES-T1100. As in those benchmarks, the Trojan leaks part of the secret key, but first XORs the leaked information with a value determined by an LFSR. When the circuit

reset signal is observed, the LFSR is loaded with the value of state. The LFSR only rotates after the Trojan has been triggered.

14. AES-T1300

a. Trigger

The Trojan_Trigger module in AES-T1300 is identical to the combinational trigger presented in AES-T700. This trigger activates on an incoming state value of 0x00112233445566778899AABBCCDDEEFF, just like the trigger in AES-T700 and AES-T1000. Note that a hash comparison of Trojan_Trigger.v in these benchmarks will fail because a single space character has been added to the version in AES-T1300.

b. Functionality

The Trojan in AES-T1300 is based on the TSC module from AES-T300. Each round key is used to drive the rotation of a shift register. The adversary can learn information about the key by using side channel analysis to observe the action of these registers. However, the shift registers in AES-T1300 will not operate until Tj_Trig = 1_2.

15. AES-T1400

a. Trigger

AES-T1400 copies its trigger mechanism from AES-T500, AES-T800, and AES-T1100. The AES-T800, and AES-T1100, and AES-T1400 versions of Trojan_Trigger.v share a common hash value of 403360F99B64A4524058DEC3268AE5AC. The Trojan in this circuit will activate after seeing the following outputs in order:

- 0x3243f6a8_885a308d_313198a2_e0370734
- 0x00112233_44556677_8899aabb_ccddeeff
- 0x0
- 0x1

b. Functionality

The Trojan in AES-T1400 is identical to that in AES-T1300. Using side channel analysis, the adversary is able to glean information about the intermediate round keys. This information results from the rotation of a group of internal shift registers. However, the registers will not rotate unless the Trojan has already been activated.

16. AES-T1500

a. Trigger

AES-T1500 uses the same counter-based trigger as AES-T900 and AES-T1200. A 128-bit counter is assigned an initial value of 0x0. Every completed encryption causes the counter to increment. When the counter's value is equal to 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF, the wire Tj_Trig will begin carrying the 1_2 signal that represents activation.

b. Functionality

The TSC module in AES-T1500 is identical to those in AES-T1300 and AES-T1400. This Trojan leaks intermediate round keys using the action of shift registers. These registers will rotate based on a combination of bits from the associated round keys and the activation of the Trojan.

17. AES-T1600

a. Trigger

The trigger in this benchmark is similar to the sequential trigger demonstrated in AES-T500 and AES-T800. However, this trigger incorporates the short-term aspect of

the AES-T400 trigger. The trigger is intended to activate after observing the appropriate input sequence, remain activated for two clock cycles, then deactivate. However, the exact HDL code provided causes the trigger to activate for one cycle on reset, regardless of inputs to the circuit. Note that the only purpose of activation in the included functionality is to load a key into a register. Leakage of the key occurs during the inactive period.

b. *Functionality*

The AM_Transmission module here is identical to that in AES-T400. Bits of the key are leaked, in reverse order, to an AM transmitter with a frequency of 1560 kHz. Each 0_2 bit is signified by one beep on this frequency, followed by a pause, while each 1_2 bit is signified by two beeps.

18. AES-T1700

a. *Trigger*

AES-T1700 uses the same counter-based trigger mechanism found in AES-T900. This trigger counts encryptions until the 128-bit counter register reaches its maximum value. After that, the Trojan is triggered. Note that there seems to be an error in the provided HDL code. Unlike AES-T400 and AES-T1600, this trigger does not include the code that would cause immediate deactivation of the trigger. Since the functionality of this circuit relies on a short-term trigger to load the AES secret key into a register, then leaks the key during the inactive period, AES-T1700 will be unable to leak the key correctly.

b. *Functionality*

The AM_Transmission module used here is very similar to the one used in AES-T400 and AES-T1600. The only source code change is the absence of the register SECRETkey, which is present in the AES-T400 module. Note that this register is unused in that module. In the AES-T1700 Trojan, bits of the key are leaked, in reverse

order, to an AM transmitter with a frequency of 1560 kHz. Each 0_2 bit is signified by one beep on this frequency, followed by a pause, while each 1_2 bit is signified by two beeps.

19. AES-T1800

a. Trigger

`AES-T1800` uses a combinational trigger that is identical to that in `AES-T700`. The Trojan is inactive at reset, then activates when the state `0x00112233445566778899AABBCCDDEEFF` is observed. Note that this trigger is not defined in a separate module, but instead is written as part of the `TSC` module. The functionality, however, is identical. The lines that define the trigger are effectively pasted into the file `TSC.v`.

b. Functionality

This Trojan performs the same battery draining operation as `AES-T500`. This is accomplished by loading a 128-bit register with alternating 0_2s and 1_2s. Once the Trojan is triggered, the register will rotate once every clock cycle

20. AES-T1900

a. Trigger

`AES-T1900` uses the counter-based trigger found in `AES-T900`, `AES-T1200`, and `AES-T1500`. The 128-bit counter will count completed encryptions until $2^{128}–1$ encryptions have been completed. After that, the Trojan will be active. A reset signal will deactivate the Trojan and reset the counter to `0x0`. Note that this trigger is included in the module `TSC`, instead of being isolated in a dedicated `Trojan_Trigger` Module.

b. Functionality

This Trojan functions the same way as `AES-T500` and `AES-T1800`. After observing the successful activation of the Trojan trigger, a register composed of alternating 1_2s and 0_2s begins rotating every clock cycle. This rotation causes an

increased power drain. In a battery-operated device, this will accelerate the failure of the device due to loss of power. Note that none of the documentation for AES-T500, AES-T1800 or AES-T1900 states exactly how much additional power drain is produced by the rotation of the DynamicPower register.

21. AES-T2000

a. Trigger

AES-T2000 uses the sequential trigger found in AES-T500, AES-T800, AES-T1100 and AES-T1400. This trigger is activated after the following values of state have been observed in order:

- 0x3243f6a8_885a308d_313198a2_e0370734
- 0x00112233_44556677_8899aabb_ccddeeff
- 0x0
- 0x1

It is not necessary for these values to be observed in sequence. The Trojan can be deactivated with a reset signal.

b. Functionality

The functionality used in this benchmark is identical to that in AES-T600. The secret key is leaked, 1 bit at a time, to a group of ten parallel wires. The adversary can read these wires through side channel analysis. Note that the key is leaked in reverse order. Each bit of the key is transmitted to this leakage circuit for $2^{128}–1$ clock cycles before the next bit is leaked.

22. AES-T2100

a. Trigger

AES-2100 is activated by the same counter-based mechanism used in AES-T900. This 128-bit counter increments with every completed encryption. When the counter contains all 1_2s, the Trojan will activate. A reset signal will deactivate the Trojan and reinitialize the counter to a value of 0x0.

b. Functionality

The functionality used in this benchmark is identical to that in AES-T600 and AES-T2000. This Trojan leaks the secret key, 1 bit at a time, to a group of ten parallel wires. The adversary can read these wires through side channel analysis. Note that the key is leaked in reverse order. Each bit of the key is transmitted to this leakage circuit for $2^{128}-1$ clock cycles before the next bit is leaked.

B. TROJANS IN BASICRSA

The second set of Trojans we will discuss applies malicious inclusions to the basicRSA circuit found on the site opencores.org. Each benchmark contains three VHDL files. The high level circuit functionality is defined in rsacypher.vhd. File modmult.vhd contains multiplication submodules for the basicRSA circuit. Note that modmult.vhd remains unchanged across all four available benchmarks, with an MD5 hash of 0x494F66BBAC36397393D842DA7A71911A. The last file in each benchmark is the test bench rsatest16.vhd. These test benches are similar to one another, but each one contains inputs specifically chosen to activate the Trojan in the benchmark circuit that contains that test bench.

Note that if you attempt to analyze these benchmarks in Vivado, it will be necessary to edit the HDL code in rsacypher.vhd. Due to a typo in this file, circuit simulation will report a value of 0xUUUUUUUU for the output cypher. This is Vivado's representation of an uninitialized value. To correct this, locate the line in process mngcount that reads:

```
elsif [sic] count = 0
```

Change this line to read:

```
elsif [sic] count /= 0
```

This change corresponds more closely to comments in the HDL, which state that this condition should be met on each round of multiplication except the first. Wire count is only 0_2 during the first round within a particular encryption.

1. Important Features of the Trojan-Free BasicRSACircuit

The basicRSA circuit is a demonstration circuit for the asymmetric RSA encryption process. Comments within the code state that this circuit is not meant for use in production environments. However, the circuit effectively demonstrates the encryption process because it uses keys small enough to be verified by external analysis.

Like other asymmetric encryption algorithms, RSA makes use of a private key, held by the keyholder, and a public key, which is available to any party who wishes to communicate with the keyholder. A communicator can use the public key to encrypt a message to the keyholder, confident that only the keyholder is capable of reading the message. Additionally, the keyholder can use the private key to sign a message, allowing communicators to verify that they did send that message.

In RSA, the two keys are composed of a total of three parts. The private key is composed of an exponent d and a modulus n. The public key is composed of an exponent e and the same modulus n. The keys can be used to convert between a plaintext message m and an encrypted ciphertext c according to the following equations:

$$c = us \pmod n$$

$$m = m^d \pmod n$$

Note that the encryption and decryption equations are structurally identical. As a result, the same basicRSA circuit can be used to perform both tasks.

See Figure 27 for a circuit diagram of the inputs to basic RSA. Bus inMod is used to input the modulus. Busses inExp and indata are used differently depending on the current circuit activity. During encryption, indata carries the plaintext message m, and inExp carries the public exponent e. During decryption, indata carries the ciphertext c and inExp carries the private exponent d. Note that the signature and verification process alters the matching of exponents to messages, but inExp is still used for the exponent and indata is used for the message to be converted.

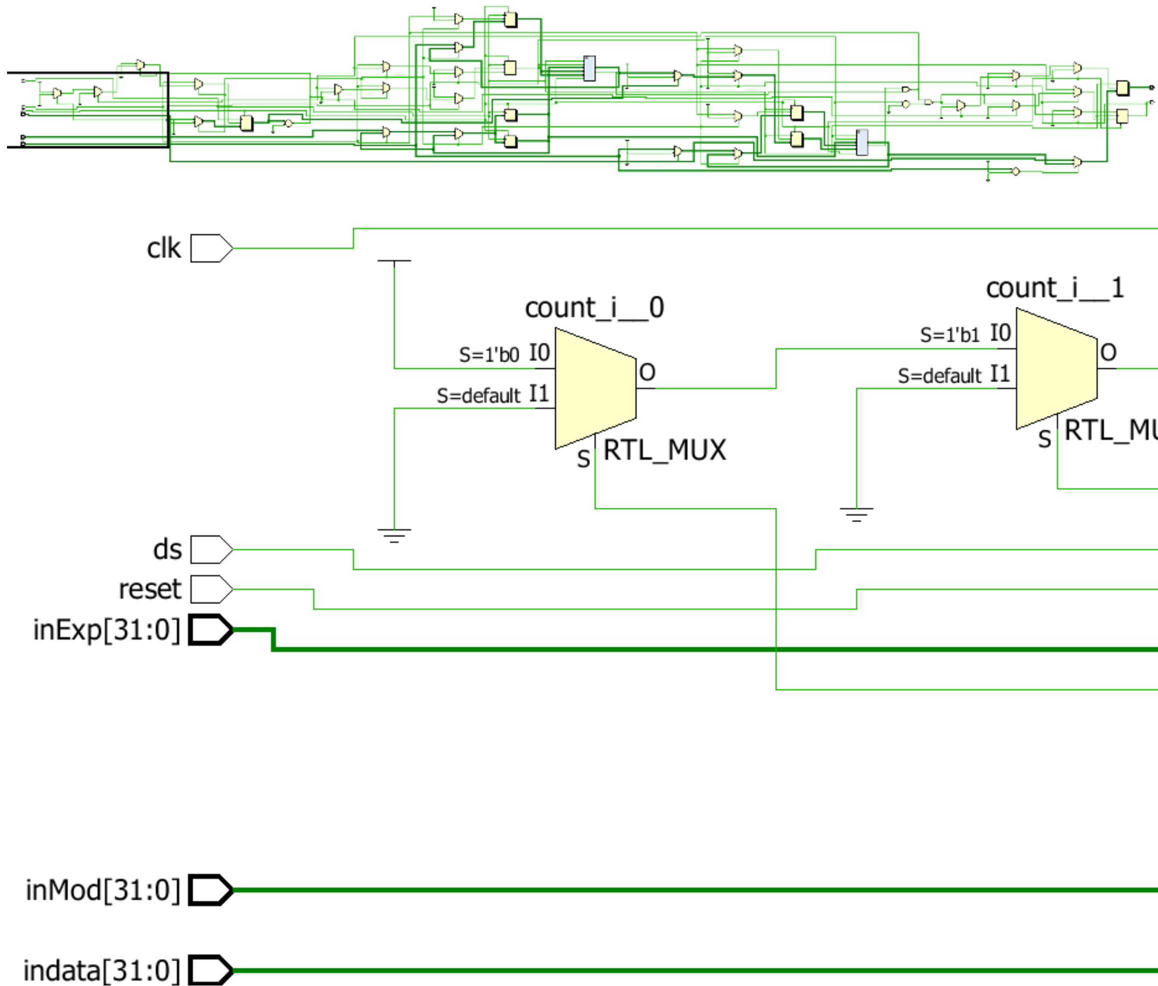


Figure 27. Inputs of the `basicRSA` circuit. The key inputs to be aware of are `inExp` and `indata`. Depending on the exact usage case of the circuit, at least one of these values is secret information not meant to be shared with the outside world. As a result, the adversary would attack confidentiality by causing the circuit to leak one or the other of these values.

The outputs of the circuit are shown in Figure 28. The primary output bus is `cypher`, which carries the final result of the RSA process. Since the RSA algorithm is the same in both directions, `cypher` can carry either the encrypted message or the plaintext message. Wire `ready` is a signal designed to alert surrounding architecture that the `basicRSA` circuit has completed the encryption and that another encryption task can be assigned.

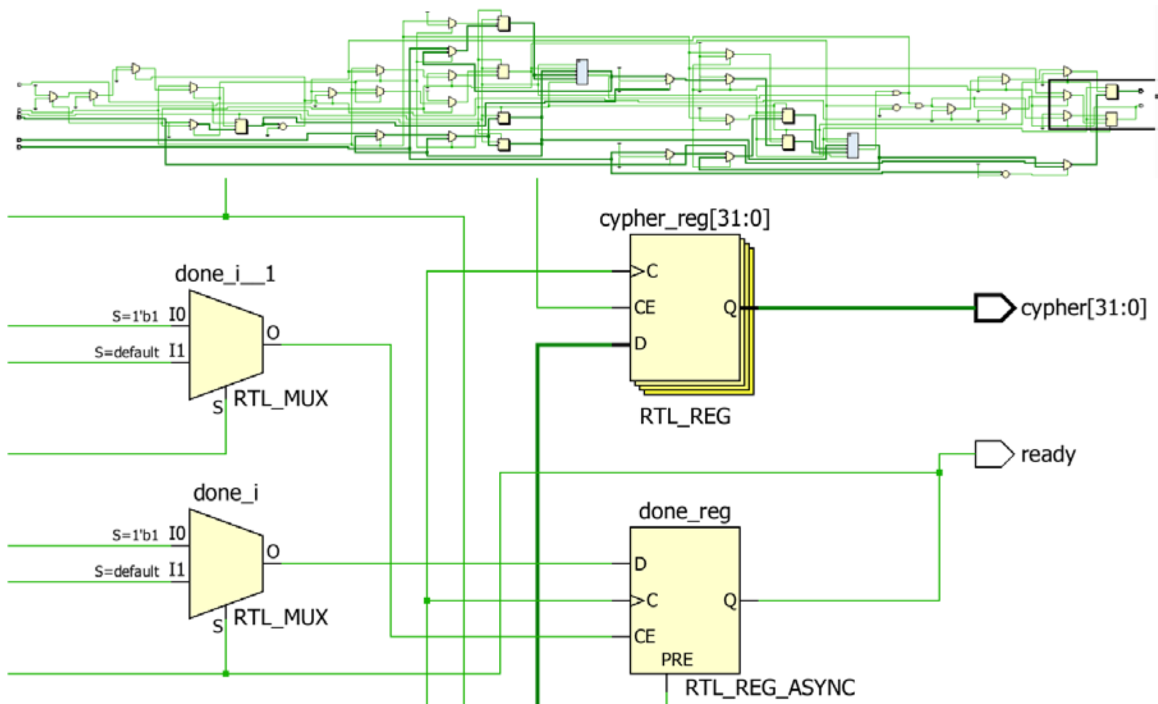


Figure 28. The outputs from the `basicRSA` circuit. Output `cypher` carries the result of the RSA operation. Due to the structure of the algorithm, this can either be an encrypted ciphertext or a decrypted plaintext. Output `ready` notifies surrounding architecture that an RSA operation has been completed.

2. BasicRSA-T100

The Trojans in the `basicRSA` benchmarks have the smallest footprint of any Trojan discussed in this thesis. As shown in Figure 29, the Trojan in `basicRSA-T100` is composed of two gates. The comparator `eqOp_i` serves as the trigger mechanism, and the trigger output is used as the select bit of mux gate `cypher_i`. This gate represents the Trojan functionality. When the Trojan is active, the output of this gate will be one selected by the adversary. The effect of this choice will be explained in the Functionality section.

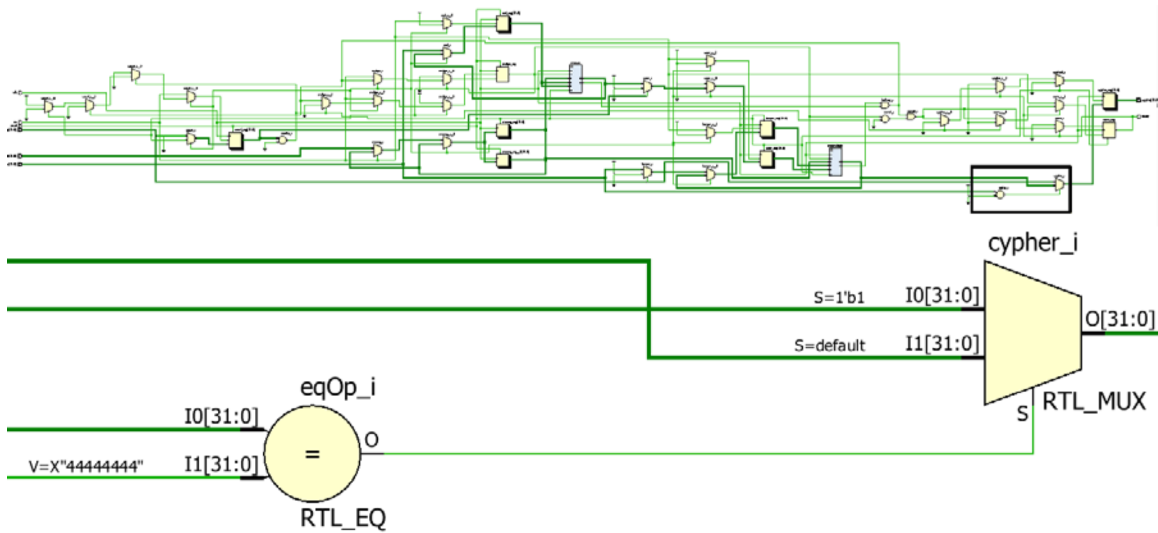


Figure 29. The complete malicious inclusion of basicRSA-T100. Comparator eqOp_i compares the input indata with 0x44444444. The result of this operation is used as the select bit for the mux gate cypher_i. The input I1 of cypher_i is the bus product, which carries the result of iterative modmult operations. When the RSA algorithm has finished, product carries the correct final result. Input I0 of cypher_i is linked directly to circuit inExp. The output of cypher_i feeds directly to output cypher.

a. Trigger

The trigger in basicRSA-T100 is a simple combinational input. As shown in Figure 29, the circuit waits for the input 0x44444444 to be sent on input indata. Vivado represents the input test as a simple equality operation between the input and the target value.

b. Functionality

The Trojan in basicRSA-T100 overwrites the output cypher with the value of inExp. Note that in the circuit diagram shown in Figure 29, this is accomplished by using the result of the input comparison trigger as the select bit for a mux gate. This Trojan is intended to be used against the keyholder of an RSA private key. Use against another communicator would allow the adversary to recover the public key, which they will already have access to. If the adversary recovers the private key, they will be able to

read any messages meant for the keyholder and also produce valid signatures in the name of that keyholder.

3. BasicRSA-T200

The Trojan in basicRSA-T200 is shown in Figure 30. Like the Trojan in basicRSA-T100, this Trojan requires only two gates. Note that these gates disrupt circuit operation at an early stage, and their effects propagate until the final output cypher is effectively replaced with the original plaintext input.

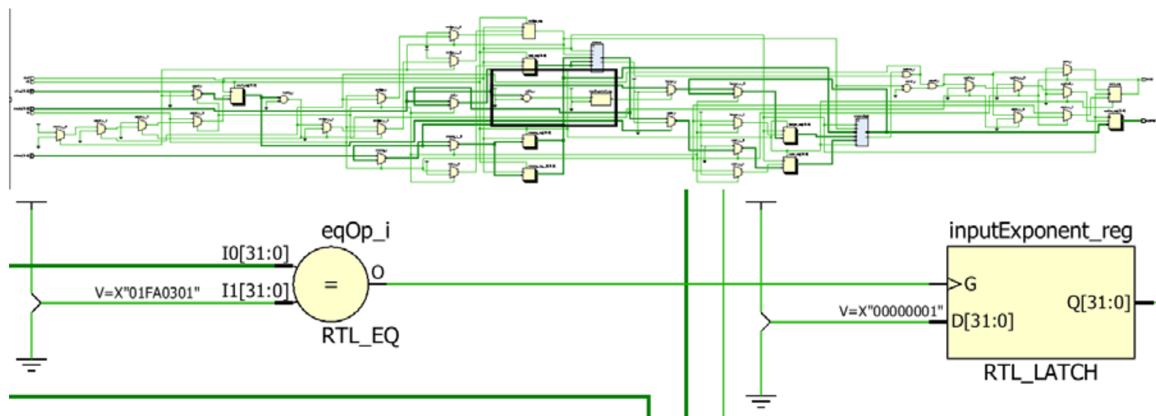


Figure 30. The complete Trojan in basicRSA-T200. Like basicRSA-T100, this benchmark triggers on a simple comparison operation. InputExponent_reg is added as an intermediate storage of inExp prior to the use of that input for the main RSA operation. If the Trojan is triggered, this intermediate storage location allows the value of inExp to be substituted with 0x00000001. The basicRSA circuit will then conduct encryption or decryption operations using the substitute value.

a. *Trigger*

The trigger in basicRSA-T200 is a simple combinational test, similar to that in basicRSA-T100. If the value of indata is 0x01fa0301, the Trojan will be triggered. As shown in Figure 30, Vivado represents this trigger with a single equality operation.

b. *Functionality*

After the appropriate input value is seen, the Trojan in `basicRSA-T200` will perform a simple replacement of `inExp`. Module `modmult` will receive the value `0x1` in place of the normal exponent.

The choice of exponent is based on the algorithm used in this circuit. The `basicRSA` that these circuits are based on uses a "square and multiply" algorithm to convert the message. For each bit of the exponent, the circuit will square the current iteration of the message. If that bit is a 1_2, then the message will be multiplied by this square, and the next iteration will be the result of this multiplication. If the bit is 0_2, then the squared value will be discarded, and the next iteration will be the same as the current iteration. The bits of the exponent are evaluated in reverse order, and a special algorithm is used to prepare the initial iteration of this process. If the least significant bit is 1_2, the message is simply loaded as the first iteration of the RSA process. If the bit is 0_2, then the value `0x1` is loaded instead. After the message has been subjected to all 32 iterations of this process, the result is transmitted as `cypher`.

The adversary selected `0x1` as the replacement value because it completely disables the encryption process. The least significant bit is 1_2, which causes the circuit to load `indata` for the initial iteration. All other bits in the attacker's chosen exponent have a value of 0_2. As a result, each iteration of the square and multiply algorithm will simply output the value of the previous iteration. After all 32 iterations are complete, the final value of `product` will be `indata`. This will be transmitted as if it were the result of the RSA algorithm. If this circuit were being used to encrypt and transmit a message, it would instead simply be transmitting the message in the clear, readable by anyone who cared to listen.

4. BasicRSA-T300

a. *Trigger*

The trigger of `basicRSA-T300` is based on a 32-bit counter with an initial value of `0x0`. This counter is shown as a register and 32-bit adder in Figure 31. The

counter increments every time the ds signal becomes 1_2. The provided test bench sets this wire to 1_2 every time a new value is inserted on the indata bus. Since this Trojan is for demonstration purposes, the creator elected to reduce the triggering value of the counter to 0x2. This means that when the second encryption begins, the Trojan functionality will immediately trigger.

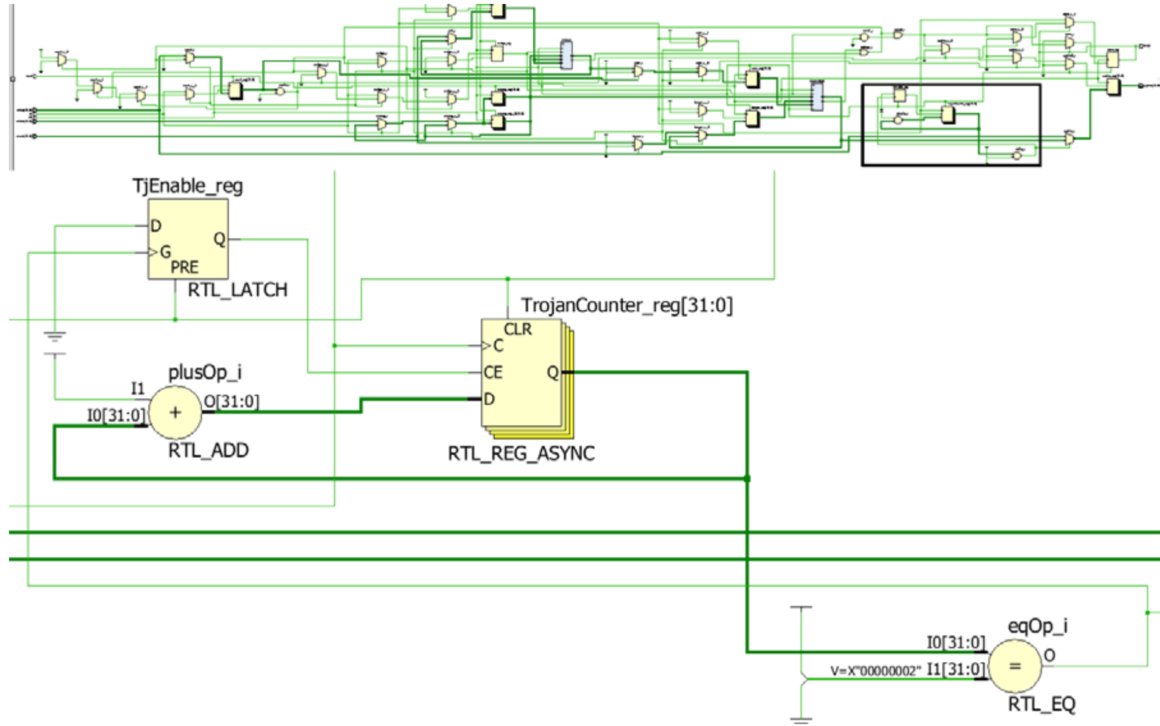


Figure 31. The trigger mechanism in basicRSA-T300. The core counter is composed of register TrojanCounter_reg and adder plusOp_i. Note that Q value of this register loops back through the adder, which adds 1 to the value on every loop. Also note that the register will only update its value if the Q value of TjEnable_reg is 1_2.

This trigger boasts one additional feature. The counter in this Trojan is tied into the internal signal TjEnable. If TjEnable carries a 1_2 signal, the counter will function as described above. If TjEnable carries a 0_2 signal, the counter will not increment. As a result, the Trojan will have a reduced impact on power consumption. This reduced consumption will interfere with side-channel detection techniques. Note that the TjEnable signal is set to 1_2 on a reset, and 0_2 when the counter has a value of

`0x2`. This means that once the Trojan has been triggered, the unnecessary counter is disabled. The only impact on the circuit operation after this point comes from the Trojan functionality itself.

b. Functionality

The functionality of `basicRSA-T300` is the same as that of `basicRSA-T100`. When the Trojan has been triggered, the cypher output will be replaced with the provided exponent `inExp`. This Trojan can capture both public and private exponents, but is most useful when used to capture a keyholder's private exponent.

5. BasicRSA-T400

a. Trigger

The trigger for `basicRSA-T400` is identical to that for `basicRSA-T300`. Using a 32-bit counter, the Trojan counts 2 encryptions, and Trojan functionality becomes active at the start of the second encryption. To reduce the Trojan's impact on power consumption and make it more difficult to detect, the counter is disabled after the Trojan is triggered.

b. Functionality

The Trojan in `basicRSA-T400` disrupts the RSA encryption by inserting an additional mux gate between `inExp` and the `modmult` module, as shown in Figure 32. Before the Trojan is triggered, the circuit will function normally. Bus `inExp` will be used to control the functionality of `modmult`, and messages will be encrypted normally. After the Trojan is triggered, this mux gate will cause `modmult` to accept the value `0x009add0a` in place of `inExp`. This changes the sequence of multiplication operations in the RSA process and produces a different final output. The resulting cyphertext has been encrypted with a key selected by the adversary. As a result, only the attacker knows the corresponding decryption key.

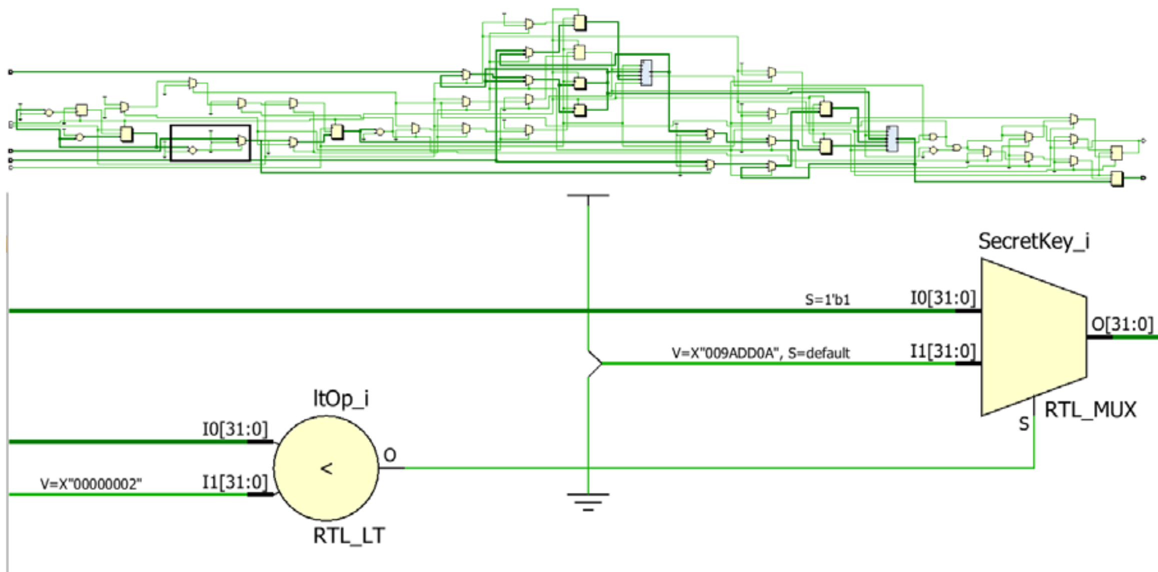


Figure 32. Circuit diagram of the functionality of `basicRSA-T400`. The comparator `ltOp_i` is the last component of the counter-based trigger mechanism. The result of this comparison will be used to select between using `inExp` to perform the RSA operation, or replacing it with the adversary's chosen exponent: `0x009ADD0A`.

Note that this attack is most useful when performed against a user who wishes to encrypt a message and send it to a keyholder. In this scenario, the intended recipient will be unable to read the message, while the adversary will. If the receiver's circuit is subverted by this Trojan during decryption, no one will be able to read the message. If the adversary uses their decryption key on the result of the attempted decryption, they will only be able to recover the original ciphertext. Without the corresponding decryption key, the adversary will remain unable to determine the original plaintext.

C. TROJANS INSERTED IN THE REGISTER TRANSFER LEVEL (RTL) OF RS232

This benchmark collection includes two major groups of RS232 benchmarks. Due to changes in the file structure and insertion stage between the two groups, this thesis will discuss each group separately. The first group, including all benchmarks from `RS232-T100` to `RS232-T901`, is based on RTL definitions of the RS232 circuit. This is the same level of abstraction that the authors used to insert Trojan functionality into the AES and RSA circuits. HDL at this level uses instructions similar to high-level software code,

including `if` statements and variables. The synthesis tools are responsible for converting this HDL to a collection of gates that acts according to the RTL specification. The other group of RS232 benchmarks, ranging from `RS232-T1000` to `RS232-T2000`, has Trojans inserted at the gate level. This HDL specifies the logical arrangement of every logic gate in the circuit. Gate-level HDL will be discussed in more detail in Section D.

Benchmarks in this set are composed of four files. File `uart.v` contains logic for the top-level RS232 circuit. This circuit instantiates a receiver module provided in `u_rec.v` and a transmitter module provided in `u_xmit.v`. All three of these files also instruct the synthesis tools to include the contents of file `inc.h`.

Note that the `include` directive in every file has a hard-coded path based on the file structure of the benchmark authors' computer. In order to synthesize these benchmarks without an error, it is necessary to change this line, which reads:

```
`include "/home/salmani_h/Trust_HUB/Trojan_Inserted/inc.h"
```

or

```
`include "/home/xuehui/project/benchmark/src/inc.h"
```

Xilinx is capable of handling relative paths in the include directive, so change the line to read:

```
`include ."/inc.h."
```

Making this change allows you to import `inc.h` as though it is another source file. The circuit will then synthesize correctly.

Note that the authors have not included a test bench for these benchmarks. We have written 3 test-benches and included them in the additional resources collection for this thesis.

1. Important Features of the Trojan-Free RS232 Circuit

The authors have not provided an explicit Trojan-free version of the RS232 circuit. However, we were able to assemble one based on the provided documentation and the use of MD5 hashes and `diff` tools. The documentation for each benchmark identifies modules affected by the Trojan. For example, the documentation for `RS232-T300` states "Trojan trigger is a 32-bit counter inserted in the transmitter part of micro-UART core." The phrase "transmitter part" refers to the `iXMIT` module, defined in `u_xmit.v`. Since the documentation did not make any reference to the `iRECEIVER` module, we decided to compare the file `u_rec.v` to the same file in `RS232-T500`, another transmitter-oriented benchmark. We discovered that the only difference between these files was the hard-coded path in the `include` directive. Since both benchmarks provide an identical `inc.h` file, this difference has no impact on the synthesis process.

I conducted similar comparisons between other files to establish baselines for each of the files. We assumed that if files of the same name remained identical across benchmarks, and none of those benchmarks documented changes to the file in question, then we could accept those files as the original, Trojan-free versions. Unfortunately, only the `inc.h` files were truly identical, with all ten versions having an MD5 hash of `0xA410FDE0B28A36ED0A13F0EB962BE60E`. For the other files, we needed to examine the `diff` results in depth. Using these results, we labeled files as functionally identical if the only differences between them were in the `include` directive or in the placement of white space. We discovered that `uart.v` remained unchanged across nine of the benchmarks, and that each of the benchmarks in this collection modified only one of the HDL files. These will be highlighted in each individual benchmark's description.

Having determined which files represented my baseline, we assembled a new Vivado project using `u_rec.v` and `u_xmit.v` from `RS232-T400`, as well as `uart.v` and `inc.h` from `RS232-T100`. We named this project `RS232-TjFree`, according to the convention followed in the AES folders.

The RS232 circuit is meant to be a sub-component of a larger architecture. This component is used to translate bytes of data into sequences of 1-bit signals that can be

sent, in sequence, from one device to another. The RS232 circuit is also responsible for converting a received sequence back to a usable byte of data. To accomplish this task, the circuit includes two nearly separate submodules, as shown previously in Figure 33. To prevent ambiguity and errors, both submodules and circuits that interact with them use certain conventions. These conventions dictate the ordering of bits in the transmitted sequence, as well as the signals to alert nearby circuits regarding when to exchange data with RS232.

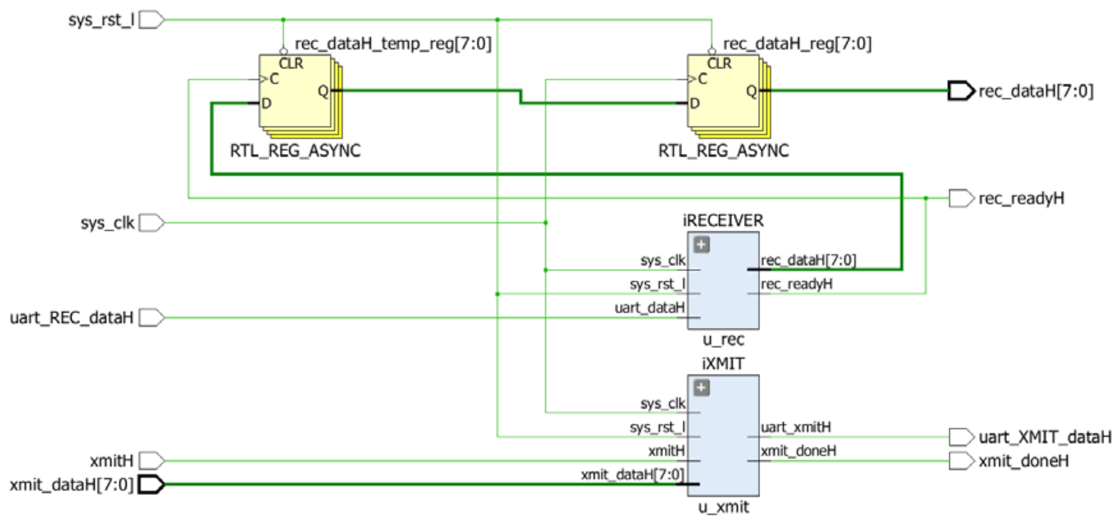


Figure 33. High level schematic of the Trojan-free version of RS232. Note that the modules iRECEIVER and iXMIT operate in near isolation. They share the same clock and reset signals, but otherwise, each has its own inputs and outputs. Module iXMIT converts the byte xmit_dataH into a series of 1-bit signals transmitted through uart_XMIT_dataH. Module iRECEIVER does the opposite, accepting a series of bits from uart_REC_dataH and converting them to the byte rec_dataH.

Figure 33 provides a black box view of the submodules iRECEIVER and iXMIT. Module iRECEIVER accepts incoming bit sequences on uart_dataH and converts them to bytes, which are sent on output bus rec_dataH. Note that a sequence is 10 bits of data. By convention, the first bit, which signals the start of the sequence, is a 0_2. The last bit is a 1_2. The 8 bits in the middle are the bits of data from this byte, ordered from least-significant to most significant. Each bit is maintained for 16 clock cycles, to

allow the receiver to read and operate on it before the next one is presented. Module iRECEIVER uses a 1-byte register to store these bits, shifting the register contents to make room for each new bit. Timing is controlled by a state machine with 5 states: r_START, r_CENTER, r_WAIT, r_SAMPLE, and r_STOP. Figure 34 shows a representation of this state machine.

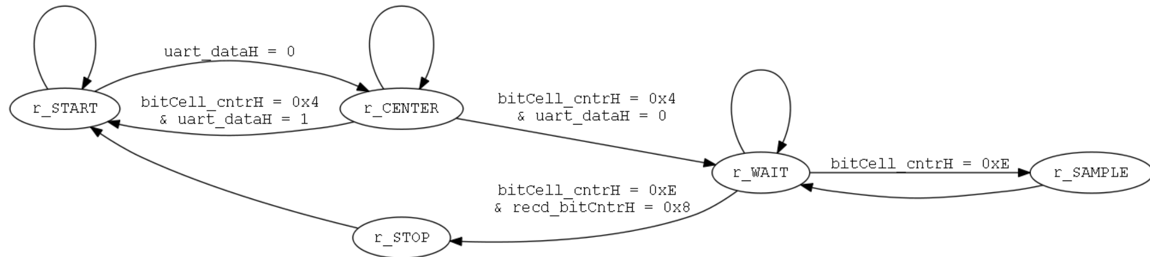


Figure 34. A diagram of the state machine in the iRECEIVER module of the RS232 circuit. Each time the machine leaves the r_SAMPLE state, a single bit is read from uart_dataH and added to the output register. State r_WAIT and register bitCellCntrH are used to control timing between each read operation. States r_START and r_CENTER confirm the initial 0_2 bit that signals the start of the message. The transition to r_STOP occurs after 8 bits have been read and another 15 clock cycles have passed. This extra time allows the circuit to account for the final bit of the serial message.

For the benchmarks discussed here, the most relevant states are r_WAIT and r_SAMPLE. The r_WAIT state will delay for 14 clock cycles, as counted by register bitCell_cntrH. On the 15th clock cycle, the machine transitions to either r_SAMPLE, where the next bit is read from uart_dataH, or r_STOP. The 16th clock cycle allows time for the state machine to transition back from r_SAMPLE to r_WAIT. A transition to r_STOP occurs when the circuit has already read the 8 bits of this byte. These bits are counted by register recd_bitCntrH. Then REC_readyH will be set to 1_2, and the state machine will transition to r_START. Be aware that some of the Trojans use the values of registers bitCell_cntrH and recd_bitCntrH as triggering conditions, so it is useful to recognize their original purpose.

After the last bit in a sequence has been observed, Module iRECEIVER will send a 1_2 signal on wire rec_readyH, signifying that data is available to be read. Note that accurate signaling on this wire is essential for proper interaction with the overarching architecture. If other circuits in the architecture read data from rec_dataH at the wrong time, they will receive a partially converted byte, and begin operating on inaccurate data.

Figure 33 also demonstrates the inputs and outputs of module iXMIT. This module accepts a byte of data on bus xmit_dataH and converts it to a 10-bit sequence according to the convention described above. This sequence is transmitted through wire uart_xmitH. To ensure that iXMIT is allowed to complete this task before a new message is provided for transmission, the wire xmit_doneH is used as an output to the external architecture. A 0_2 signal on this wire signifies that the iXMIT module is still in the process of sending the current byte. If a new input is applied to the xmit_dataH bus, the resulting sequence will be part of the current byte and part of the new byte. Ignoring this convention would prevent any decipherable message from being sent.

To control the transmission and timing of each byte, iXMIT uses a state machine similar to that in iRECEIVER. The states in this module are x_IDLE, x_START, x_WAIT, x_SHIFT, and x_STOP. In state x_SHIFT, the next bit is transmitted on uart_xmitH, and the bit count is incremented. State x_STOP represents the completion of the transmitter conversion process. The Trojans written for the transmitter have less direct interaction with this state machine than the Trojans written for the receiver module have with their state machine.

Note that iRECEIVER and iXMIT share the inputs sys_clk and sys_rst_l. wire sys_clk[sic] is the system clock, used for synchronization purposes. The sys_rst_l input is the reset input, which allows the circuit to be returned to a pre-activation state. Note that this reset works differently than the rst wire used in the AES benchmarks. The RS232 circuits are inactive when sys_rst_l carries a 0_2 signal. To send messages, it is necessary to load this input with a sustained 1_2 signal. A system reset involves setting the sys_rst_l value to 0_2, then back to 1_2.

When this thesis discusses the Trojan functionality contained within each benchmark, we assume that the surrounding architecture follows the conventions described above. Several of these Trojans interfere with the wires `rec_readyH` and `xmit_doneH`, which control when nearby circuits attempt to send or collect data. To fully explain the impact of these Trojans, it is necessary to assume that linked circuits conform to expected conventions.

2. RS232-T100

The Trojan in `RS232-T100` affects only the `iRECEIVER` module. All modifications to the HDL code of this benchmark occur in the file `u_rec.v`. Files `uart.v` and `u_xmit.v` are functionally identical to files of the same name found in other benchmarks.

a. Trigger

The documentation of RS232 states that the Trojan is triggered by a comparator across 19 signals. The documentation does not identify those signals. After searching the HDL code and the schematics, we were able to discern that these 19 signals are the individual bits of the registers `bitCell_cntrH`, `recd_bitCntrH`, and `state`, and the output `rec_dataH`. The relevant registers are shown in Figure 35, along with the first AND gates involved in funneling all 19 values into a single wire that represents the triggered state of the Trojan. Register `state` is the `iRECEIVER` module's mechanism for identifying the current state of the internal state machine. Note that this means that the Trojan will be triggered at the end of the receiver's conversion process. The Trojan will trigger when the following combination of values is observed:

- `recd_bitCntrH = 0x3` (3 bits of the key have been read.)
- `rec_dataH=0xFF` (This register contains some bits from the current message and some from the previous message. Each of these bits is 1_2.)
- `state = 0x3` (The circuit is in the `r_WAIT` state.)

- `bitCell_cntrH = 0xE` (The circuit will be transitioning to the `r_SAMPLE` state during this clock cycle.)

Note that the value of `recd_bitCntrH` is determined by the number of bits that have been read from a given input. This means that the `rec_dataH` value is not based on a single input, but rather the partial combination of two successive inputs. In particular, the first 5 bits of the initial input must be 11111_2, and the last 3 bits of the current input must be 111_2. This causes the register to contain 11111111_2 when `recd_bitCntrH` is `0x3`. Register `bitCell_cntrH` is guaranteed to cycle to `0xE` once for each bit read, and the value of `state` corresponds to `r_WAIT`, which is part of the reading process.

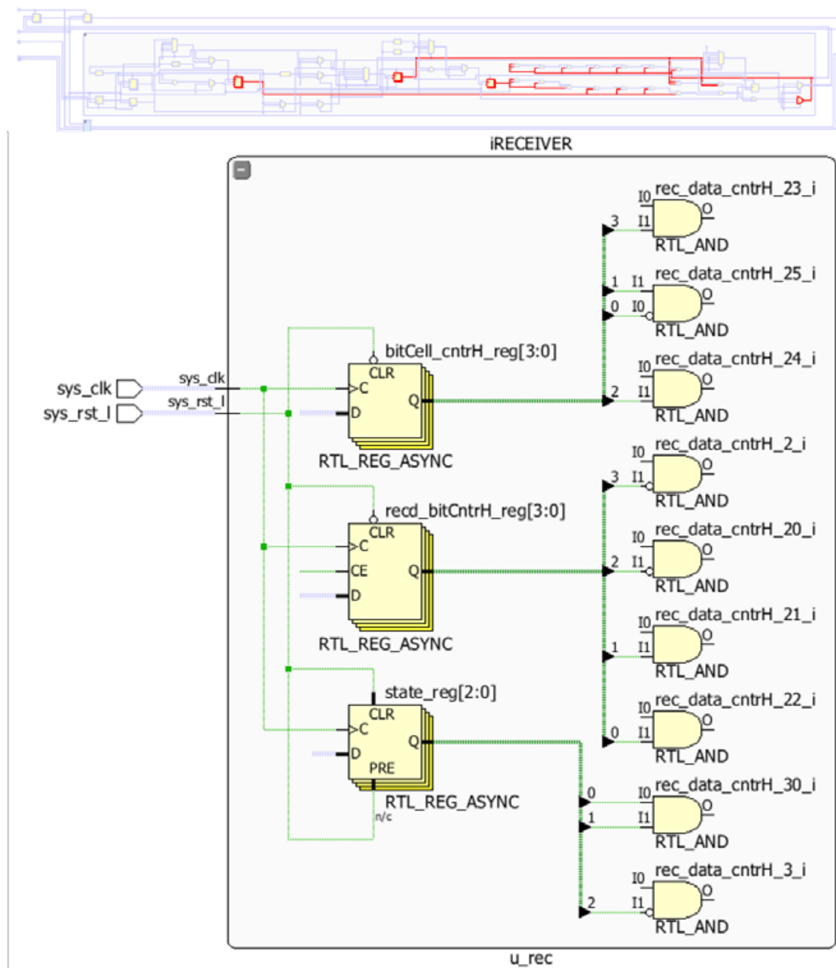


Figure 35. Partial schematic showing the trigger registers of `RS232-T100`. These registers, in conjunction with the current value of `rec_dataH`, form the 19-bit trigger value of this Trojan. AND gates are used to funnel these inputs into a single result wire labelled `ena`. The world view above highlights the registers, showing how Vivado places them in the schematic.

b. *Functionality*

The Trojan in this circuit is a simple mux gate, which accepts the final trigger result as the select bit. This gate selects between the correct result of output `rec_readyH` and a signal of 0_2. This means that prior to Trojan activation, `rec_readyH` will be 1_2 when the `iRECEIVER` circuit has a complete byte of data ready to be read, but when the Trojan is active, the `rec_readyH` signal will always be 0_2. The external architecture will never be notified that there is data available to collect. A second mux gate selects between the correct `rec_dataH` result and `0x00`. Note that the documentation only discusses the interference with `rec_readyH`. The additional overwrite of the converted data prevents the surrounding architecture from reading any meaningful answer, even if that architecture is designed to ignore the conventions of `rec_readyH`. Both mux gates are shown in Figure 36.

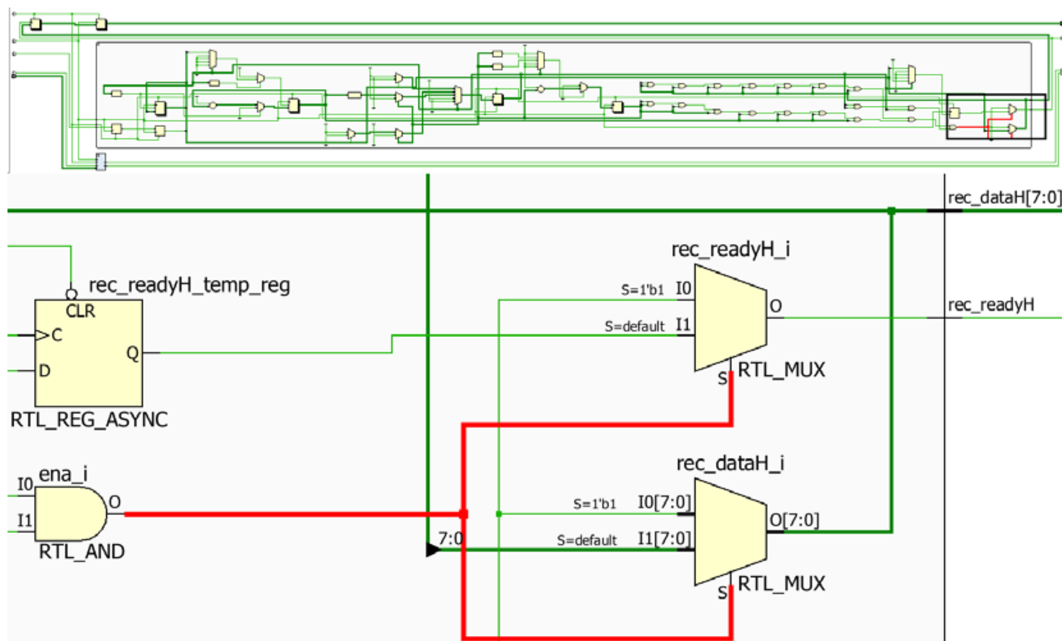


Figure 36. Schematic of the gates controlling the functionality of `RS232-T100`. The AND gate `ena_i` is the last gate in the trigger comparison process. The result of this operation is used as the select bit for two mux gates. One gate determines the output `rec_readyH`, and the other determines the output `rec_dataH`. Note that the result of the mux gate rec_dataH_i is fed back to another area of the receiver sub-circuit. This allows it to be used as part of the triggering conditional.

3. RS232-T200

Like RS232-T100, this benchmark affects only the iRECEIVER module. iXMIT will continue to function normally. The Trojan triggers based on a set combination of values on two internal registers and the module output. After activation, this Trojan activates a counter. This counter has almost no impact on the circuit.

a. Trigger

The documentation for the benchmark describes the trigger as a comparator across 16 signals, but fails to identify the signals. The signals, like those in RS232-T100 are based on internal registers and the output rec_dataH. Note that the register state is not part of the triggering condition in this trigger. According to the documentation, the required combination of values to trigger this Trojan is as follows:

- recd_bitCntrH = 0xF
- rec_dataH=0xFF
- bitCell_cntrH = 0xF

Note that as written, it is impossible to activate this Trojan. According to the HDL code, register recd_bitCntrH will count from 0x0 to 0x8 during the process of converting a single byte. When the circuit prepares to read the next byte, it will reset recd_bitCntrH to 0x0. The register will never hold a value of 0xF. In order to simulate this Trojan, we changed the HDL code to trigger the Trojan when this register held a value of 0x3, based on the example of RS232-T100. The relevant line in RS232-T200 is:

```
Assign […] recd_bitCntrH[0]&recd_bitCntrH[1]&recd_bitCntrH[2]&recd_bitCntrH[3];
```

The equivalent line in RS232-T100 reads:

```
Assign[…]recd_bitCntrH[0]&recd_bitCntrH[1]&(~recd_bitCntrH[2]&(~recd_bitCntrH[3]);
```

b. Functionality

The documentation claims that this Trojan reduces design reliability through the use of a counter. The HDL does include code to create a counter, which only increments

when the Trojan has been triggered. Note that this counter is never used as an input to any other part of the receiver sub-module. Because of this, Vivado will not include this counter or the trigger mechanism in the synthesized or elaborated designs. To produce the schematic in Figure 37, we added an output to the definition of the receiver module, and directly assigned `count_l` as the source of that output. With the output included, Vivado will actually display the entire malicious inclusion in the schematics and assign values to Trojan elements in the simulation waveform. Note that the counter still does not leak information or interfere with any other part of the receiver.

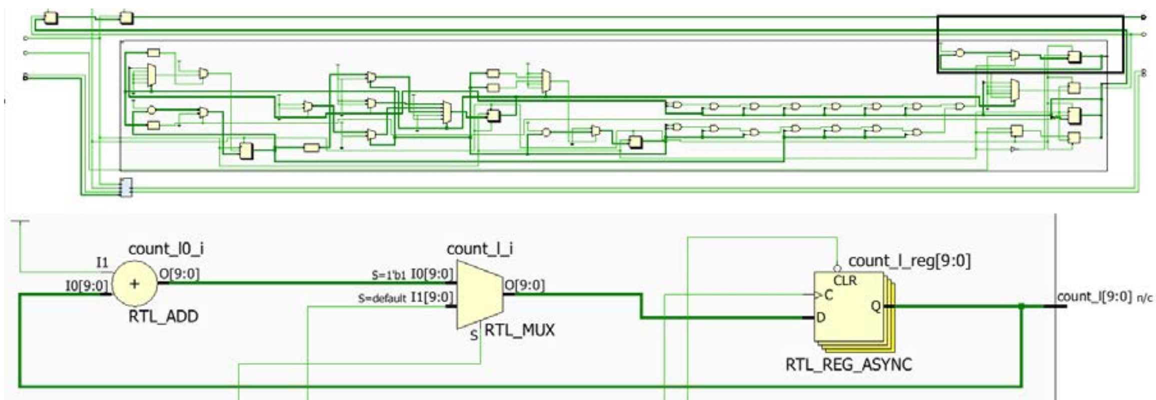


Figure 37. The Trojan in `RS232-T200`. The three gates shown make up a 10-bit counter, which increments every cycle while the Trojan is active. Note that this counter is not used as an input to any other part of the circuit. Also note that output `count_l` was artificially added to prevent Vivado's automatic optimization from excising both the Trojan and the trigger.

4. RS232-T300

The RS232-T300 benchmark demonstrates Trojan functionality in the transmitter module of the RS232 circuit. In this benchmark, `u_xmit.v` is the modified file. Files `uart.v` and `u_rec.v` are functionally identical to those we used in the Trojan-free circuit.

a. Trigger

This benchmark uses a 32-bit counter as a Trojan. This counter, shown in Figure 38, begins with a value of `0x0`, and increments every time a byte of data is transmitted. Note that the Trojan functionality in this circuit will not trigger until this counter reaches a value of `0xFFFFFFFF`. In order to simulate this Trojan, we found it necessary to edit the HDL code in `u_xmit.v`. We changed the starting value of the counter to `0xFFFFFFFC`, which caused the Trojan to trigger after four transmissions. Note that after activation, the counter resets to the initial value, but the Trojan remains active until a reset signal is observed.

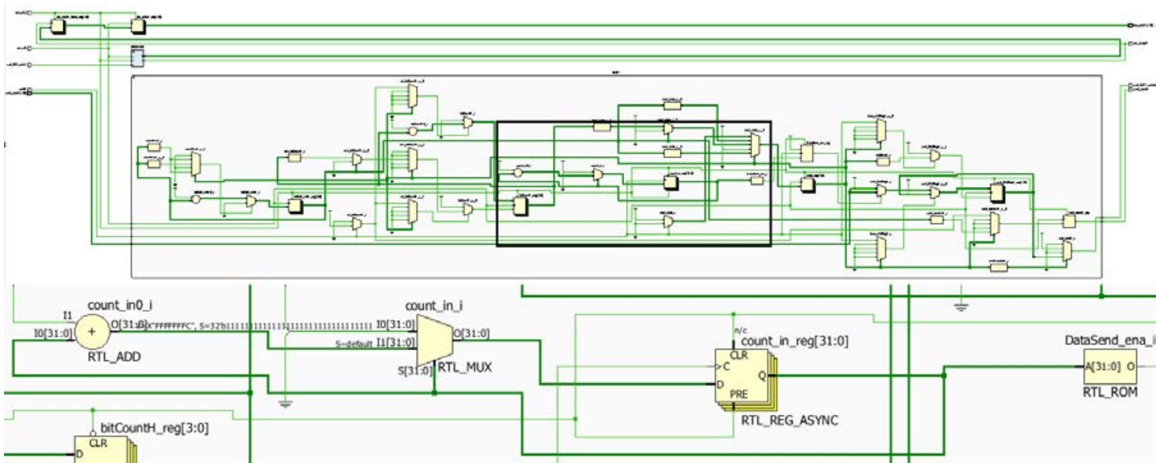


Figure 38. Partial Schematic of the trigger mechanism in `RS232-T300`. Register `count_in_reg`, the adder, and the mux gate `count_in_l` form a counter that is designed to count from `0` to `0xFFFFFFFF`. When the counter reaches the final value, the ROM unit `DataSend_ena_reg` will send a 1_2 signal, representing activation.

b. Functionality

The documentation states that the Trojan will replace the 7th bit of every message transmitted after activation. After some experimentation, we have concluded that this statement refers to bit `7`, assuming that the least-significant bit is numbered `0`. Thus, the highest order-bit of each byte is set to 1_2, regardless of its original value. Note that some messages sent by `RS232-T300` will be unaffected because the highest-order bit already

is 1_2. For example, the byte 0x4C will become 0xCC, but the byte 0xA2 will still be transmitted as 0xA2.

5. RS232-T400

The Trojan in RS232-T400 does not affect the internal workings of either of the submodules in the circuit. Files u_rec.v and u_xmit.v are unaltered. Instead, the authors implemented this inclusion in the file uart.v.

a. Trigger

The Trojan in this benchmark uses a comparison-based trigger, as shown in Figure 39. This trigger tests the values of rec_dataH and xmit_dataH against 0x4C. If both busses hold this value at the same time, the Trojan will trigger. Note that the trigger is not saved. When one of these wires no longer holds a 0x4C value, the Trojan will cease to operate. Note that the HDL code doesn't directly refer to the value 0x4C. The value is assembled by concatenating X_START, X_WAIT, and X_SHIFT, which are all values defined in inc.h. The relevant line is:

```
if((rec_dataH_rec==xmit_dataH)=={x_START,x_WAIT,x_SHIFT[1:0]})
```

The trigger does not sustain its effect. When either of the triggering inputs changes to a value other that 0x4C, the Trojan will deactivate the next time xmit_doneH is 1_2.

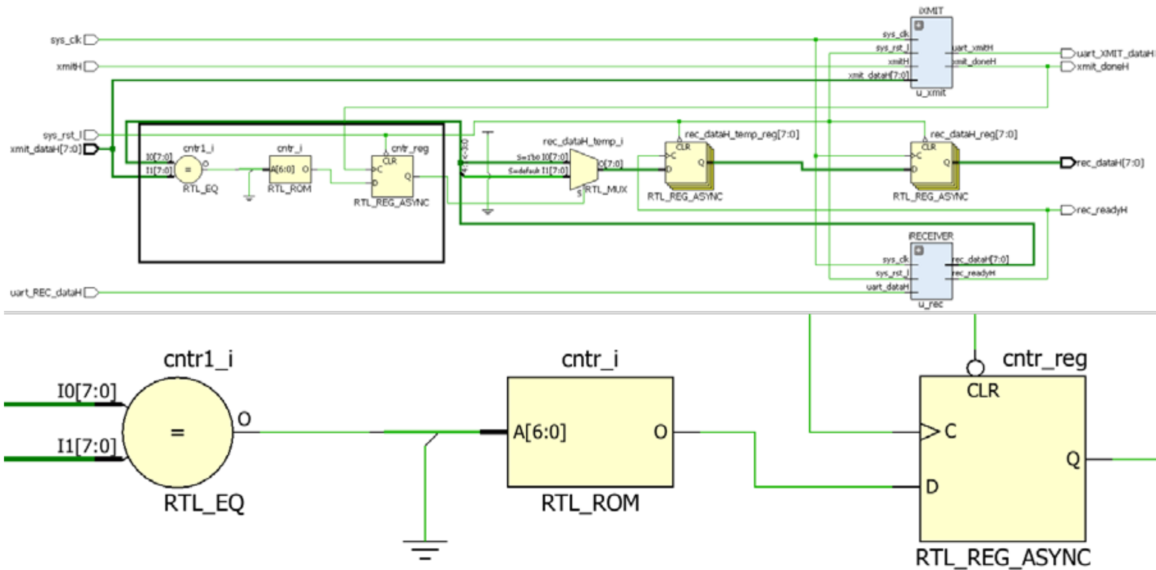


Figure 39. The trigger mechanism of RS232-T400. The `RTL_EQ` primitive is responsible for comparing `rec_dataH` against `xmit_dataH`. The result of this operation is fed into ROM `cntr_i`. If the values are equal, the value of register `cntr_reg` will be set to 1_2. This wire is used as a select bit for a mux gate, which controls the final `rec_dataH` output from the circuit.

b. Functionality

This Trojan partially replaces the data of the output rec_dataH before that data is transmitted to the rest of the circuit. The replacement is accomplished by shifting all of the bits in the lower nibble by one place. The least significant bit in the upper nibble is simply overwritten. A 1_2 bit is appended to this value to produce a full byte of data. This process transforms the triggering value of `0x4C` into a transmitted value of `0x59`. Note that since the trigger is only active for the duration of the `0x4C` input, this is the only value that will be converted.

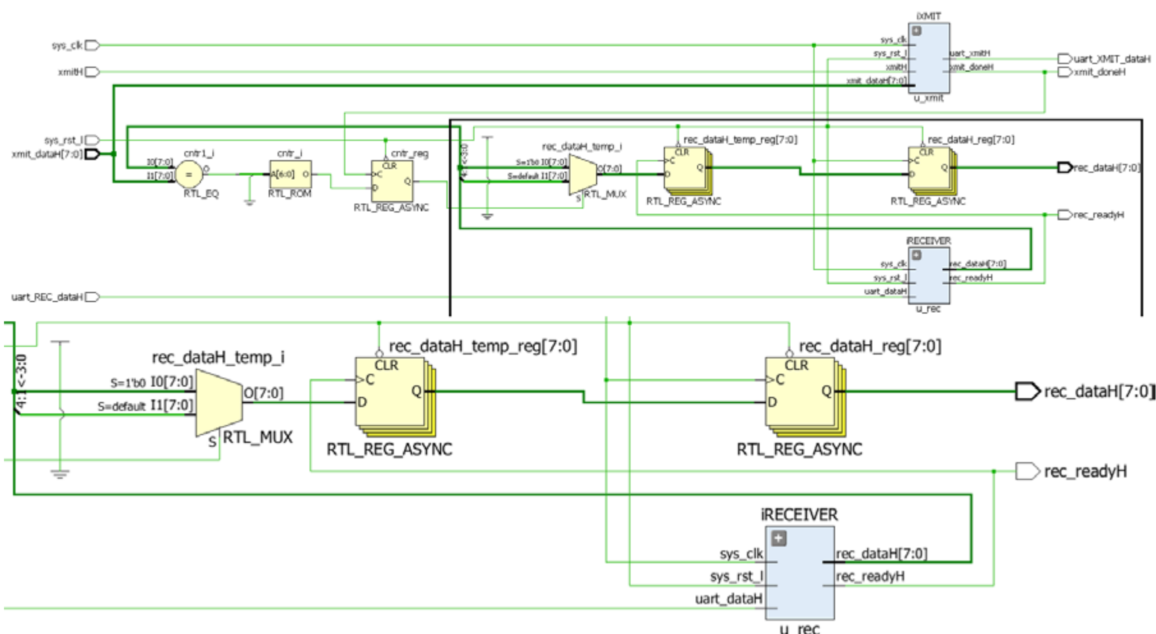


Figure 40. The Trojan functionality of RS232-T400. The key feature of this Trojan is the mux gate `rec_dataH_temp_i`, which is used to determine whether the final circuit output should be the correct value of `rec_dataH`, as determined by the `iRECEIVER` module, or a reordered combination of bits. Note that both `iRECEIVER` and the overall circuit have an output labelled `rec_dataH`. In every other circuit in this group, the distinction is unnecessary because the module output is fed to the overall circuit output without modification.

6. RS232-T500

a. Trigger

The module `RS232-T500` uses the same trigger mechanism as that in `RS232-T300`. A 32-bit counter is incremented after every complete transmission from the `iXMIT` module, and after the counter has reached the maximum value, the Trojan triggers. Note that in order to effectively analyze this Trojan, we alter the HDL code in `u_xmit.v`. Based on my modification, the counter's initial value is `0xFFFFFFFC`. Note that after the Trojan triggers, the counter will return to its initial value, but the Trojan will remain activated until the circuit is reset.

b. Functionality

After the Trojan in the circuit is activated, the signal xmit_doneH will be stuck at 0_2. This signal prevents the external circuit from recognizing that the iXMIT module is available to transmit more data. The result is a straightforward denial-of-service. Note that an external architecture could bypass this by sending data to the circuit anyway. The transmitter would still run through the process of parsing and transmitting the byte. However, the xmit_doneH convention provides a lock mechanism and reduces race conditions. Without it, the delineation between messages could be disrupted by other circuit activity.

7. RS232-T600

a. Trigger

The trigger is activated by a sequence of inputs on xmit_dataH. If the following sequence is observed, the Trojan will be triggered: 0xAA, 0x55, 0x22, 0xFF. Note that these inputs must be observed in order and in immediate succession. To control this, the authors implemented a state machine, shown in Figure 41. Each state represents a certain stage in the sequence. If the correct value is observed on xmit_dataH, the machine will transition to the next state. The appearance of another value in the middle of the sequence will cause the circuit to discard all previously observed sequence values and start again. Also note that the documentation incorrectly identifies the sequence as 0xAA, 0x55, 0x00, 0xFF. The value 0x22 can be found in the HDL code, and we have confirmed in simulation that this sequence triggers the Trojan. After the Trojan is triggered, it will remain active until a reset occurs.

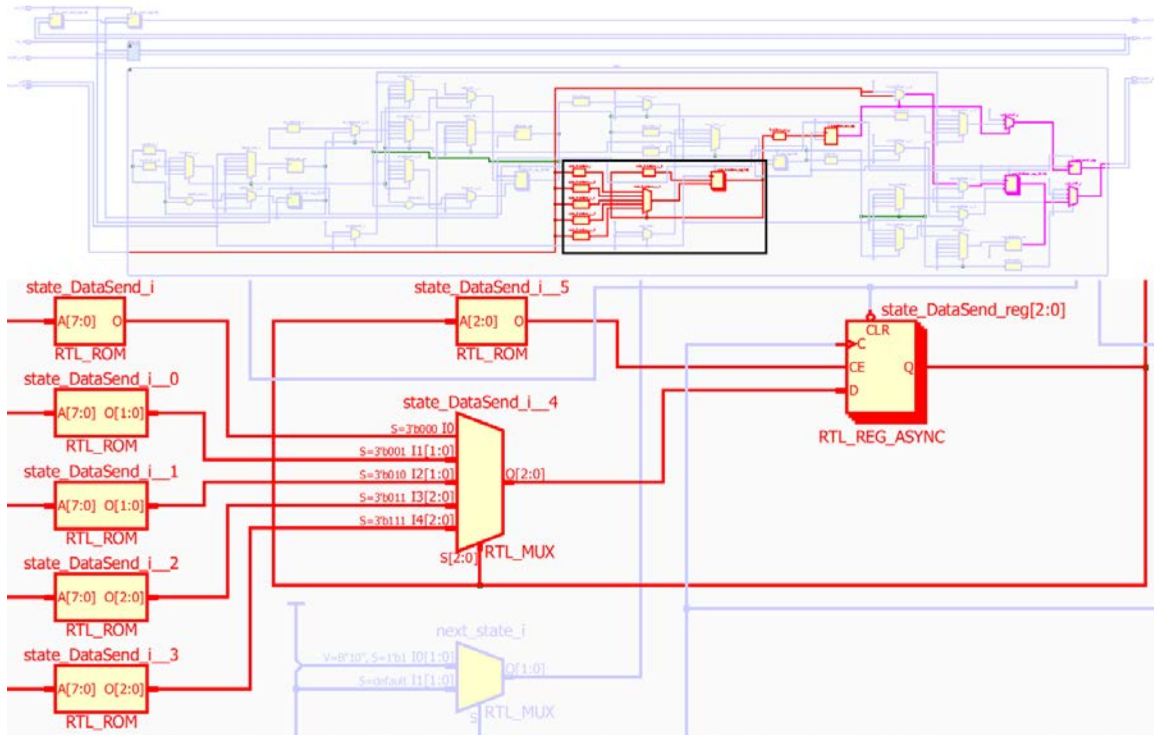


Figure 41. The RTL layout of the state machine that controls the trigger of RS232-T600. Each state_DataSend_i__# ROM module represents a potential state. The output of that module is dependent on the current value of xmit_dataH. The mux gate state_DataSend_i__4 will only select one of these values to pass through to state_DataSend_reg. This register actually contains the select bits responsible for that choice; the current state is responsible for determining which values can potentially be passed to the register.

b. Functionality

After the Trojan is triggered, the outputs of iXMIT will both be affected. Output xmit_doneH will be set to 1_2, ensuring that this circuit announces that it is always ready to transmit a new message. This means that each byte will be sent to iXMIT as soon as it exists. If the external architecture wishes to send a lot of data, then the iXMIT byte could change every clock cycle. Each bit transmitted could come from a different message. When the circuit does receive a new message to transmit, it will replace the high-order bit of the message with a 1_2. When we ran this in simulation using my test-bench, bytes 0x22, 0x17, 0x11, 0xBA, and 0x41 were transformed to 0xA2, 0x97,

0x91, 0xBA, and 0xC1, respectively. Note that byte 0xBA was essentially unaltered. Also note that my test-bench was written according to transmission timing, not according to the convention.

8. RS232-T700

a. Trigger

The trigger in RS232-T700 is identical to that in RS232-T600. The Trojan will be activated after xmit_dataH carries the following bytes, in immediate succession: 0xAA, 0x55, 0x00, 0xFF.

b. Functionality

Module RS232-T700 borrows its functionality from RS232-T500. The xmit_doneH signal is stuck at 0_2 from activation until the next reset. As long as the external architecture obeys the condition, iXMIT will never receive another signal to transmit.

9. RS232-T800

The RS232-T800 benchmark was constructed by modifying the HDL present in u_rec.v. Files uart.v and u_xmit.v are unchanged.

a. Trigger

The trigger is borrowed from RS232-T100. It triggers based on the presence of the following register and input values:

- recd_bitCntrH = 0x3
- rec_dataH=0xFE
- bitCell_cntrH = 0x7
- state = 0x3

The trigger is written in such a way that the Trojan will deactivate as soon as any part of the triggering state changes. This will happen in a single clock cycle, when register `bitCell_cntrH` increments.

b. Functionality

The functionality of this circuit produces a subtle error in the interpretation of messages. When the Trojan in this circuit is triggered, `rec_readyH` will be set to 1_2 immediately. Note that `recd_bitCntrH` has a value of `0x3`. This means that `iRECEIVER` has not finished converting the current byte. Output `rec_dataH` is carrying a value constructed from part of the current byte and part of the previous byte, but the `rec_readyH` signal of 1_2 signifies that data is ready for collection.

The external architecture has been designed to accommodate this convention, and the byte `0xFE` will be immediately collected and subject to whatever operations were planned for the next byte. The problem becomes worse after 5 more bits have been read. Now, `iRECEIVER` actually has finished reading a byte, and `rec_readyH` will be set to 1_2 again. The architecture will accept this new value and all subsequent values as additional data. A byte will have been inserted into the sequence, but everything will otherwise be in order.

The impact of this depends heavily on the purpose of the overall architecture. For example, if this RS232 module is accepting a private RSA key from a card reader, that key is now incorrect. Any signature computed using that key cannot be verified using the public key.

10. RS232-T900

The benchmark in `RS232-T900` was created by modifying the HDL code in `u_xmit.v`. Specifically, the authors added an extra state to the existing `iXMIT` state machine and defined a process to recognize a sequence of inputs coming into the `iXMIT` module.

a. Trigger

The `RS232-T900` trigger is similar to the trigger mechanism used by `RS232-T600` and `RS232-T700`. In this benchmark, the Trojan will trigger after the following sequence of inputs is encountered in immediate succession: `0xAA`, `0x55`, `0x22`, `0xFF`. Note that the documentation for `RS232-T900` incorrectly states that the input sequence is `0xAA`, `0x55`, `0x00`, `0xFF`. The value `0x22` is the third value listed in the HDL code, and we were able to simulate this Trojan's activation using that value instead of `0x00`.

This trigger uses a simple state machine to track how many of the sequence values it has seen, and which value it should expect next. Note that if the `iXMIT` module receives any value other than the expected value, the state machine will immediately revert to the starting state, and the expected value will be set to the first value in the sequence.

b. Functionality

Module `RS232-T900` uses a denial-of-service Trojan, designed to lock the circuit in a non-transmitting state. To accomplish this, an extra state is added to the transmitter's internal state machine.

11. RS232-T901

This benchmark is not a distinct Trojan. It is a new version of `RS232-T900`. The only distinction between these two benchmarks is the combination used to trigger the Trojan.

a. Trigger

In `RS232-T901`, the Trojan will trigger when the following sequence of inputs is observed on wire `xmit_dataH`: `0xAA`, `0x00`, `0x55`, `0xFF`. Note that these inputs must be observed in order and in immediate succession. The trigger has no long-term mechanism for remembering its place in the input sequence.

b. *Functionality*

Module `RS232-T901` uses the same denial-of-service mechanism discussed in `RS232-T900`. When the Trojan is triggered, the `iXMIT` module enters the `x_DataSend` state. The circuit will only transition out of this state at circuit reset. While in `x_DataSend`, the circuit ignores inputs and makes no changes to the outputs.

D. TROJANS INSERTED IN THE GATE LEVEL OF RS232

The final set of benchmarks demonstrates the insertion of Trojans in gate-level HDL code. Like those discussed in the previous section, these Trojans target an RS232 implementation. This implementation follows a different file structure than that observed previously. Instead of a modularized design composed of three files, each benchmark contains two versions of the file `uart.v`. One of these is designed for 90 nm circuits, and the other is designed for 180 nm circuits.

For the purposes of this thesis, we will concentrate our discussion on the 180 nm version of each benchmark. This decision is based on the fact that several elements of the provided documentation for each benchmark are drawn from this version and directly contradict the HDL code provided for the 90 nm version. In particular, each benchmark's documentation actually includes a small segment of Verilog code. This code, which defines the malicious inclusion, is directly copied from the 180 nm version of the circuit. Using `find` operations, we have confirmed that the documentation does not precisely reflect the 90 nm version of `uart.v` in any of the following benchmarks. At least one of the documentation-defined gates is missing in every case. In several cases, the 90nm Trojan does not have any capacity to control output signals that the 180nm Trojan has direct access to. In order to provide a clear, consistent picture of each Trojan, this thesis will concentrate on the inclusions reflected in the provided documentation.

Note that none of the provided benchmarks will synthesize in Vivado. These benchmarks were produced using Synopsys standard libraries, which are not included by default in Vivado. As a result, any attempt to synthesize `uart.v` alone will fail. Vivado will state that modules cannot be instantiated because they are undefined. To fix this, we

wrote a Verilog library containing definitions for each of the missing modules. To implement these modules correctly, we used the documentation for each library, which includes a truth table describing the behavior. We would like to thank Dr. Wei ("Vinnie") Hu of UCSD for providing starter code for a selection of the Synopsys modules. This starter code provided a model that allowed us to implement each of the necessary modules. Note that we have provided two separate libraries in the software artifacts for this thesis. The first, `lib_90.v`, will allow you to synthesize the 90 nm version of each benchmark. The second, `lib_180.v`, was written for the 180 nm version. In either case, import the library file to your project as if it were a standard Verilog file. With this file included, Vivado will be able to run elaboration, synthesis and simulation without reporting an error.

While using the library will allow Vivado to synthesize each benchmark, you will discover that it does not synthesize correctly. Vivado's synthesis tool performs many optimization steps, which prune redundant or non-transitive logic from each circuit. In the RS232 circuits discussed in this section, Vivado will optimize away most of the circuit, including the entire inclusion. We have been unable to completely disable optimization. However, we discovered that the elaborated design and the behavioral simulation are not impacted by Vivado's optimization. Once you have established that your circuit synthesizes without Vivado reporting an error, you will find it easier to use the pre-synthesis design for evaluation of each benchmark. If you wish to perform post-synthesis studies, you can reduce the impact of optimization in the synthesis settings dialog. Change the setting "flatten hierarchy" to "none." This will prevent Vivado from optimizing away some of the modules listed in the HDL code, though it does not protect every module.

1. Common Features of the Gate Level RS232 Benchmarks

The authors do not provide a Trojan-free implementation of this circuit, and we have been unable to locate a definite source for the HDL code provided to represent these circuits.

Due the lack of a Trojan-free implementation, we will use this section to discuss the common features of these benchmarks, including inputs, outputs, and registers. We will also provide a brief description of the structure of the malicious inclusion used in this set. We include this discussion here because, with the exception of `RS232-T1800`, the inclusions in this set are variations on a common structure. Later explanations will be more easily explained in terms of this common structure.

This version of the `RS232` module does not list explicit modules for the receiver and the transmitter, but, it does retain the same set of inputs and outputs as the Trojan-free RTL circuit shown in the previous section. Input `uart_REC_dataH` accepts a sequence of bits, which is converted to the byte `rec_dataH` and provided to the external architecture. Output `rec_readyH` signals the completion of this process. The 8-bit input bus `xmit_dataH` accepts data from the architecture and converts it to a sequence of bits. Output `xmit_doneH` is used to inform the external architecture that the circuit is ready to accept and transmit another message. The bit sequences follow the same convention used in the previous `RS232` circuits. The transmitter sends a 0_2 to signal the start of a new message, sends each bit in order from least-significant to most significant, and ends the sequence with a 1_2.

In addition to inputs and outputs, the gate-level implementation of module `RS232` seems to use the same registers as the RTL version of the circuit. Note that the gate-level implementations of module `RS232` all include a group of flip-flops with the names `iXMIT_state_reg_0_`, `iXMIT_state_reg_1_`, and `iXMIT_state_reg_2`. These flip-flops seem to represent the 3-bit register `state` from the `iXMIT` module of the RTL version of this circuit. Other registers from the RTL circuit also seem to be represented by flip-flops in the gate-level circuits. This correlation allows us to discuss the circuit in terms of multi-bit register activity, instead of only discussing the value of individual bits. This will be particularly significant when discussing the activities of counters like the transmitter's `bitCountH` register.

Figure 42 showcases the structure of a typical inclusion from this set. The exact layout varies across benchmarks, but each inclusion includes a single gate that marks the

transition between trigger and functionality. In almost every case, this gate is an OR gate labeled U302. The Y output of this gate, identified as iCTRL in the HDL code, is the centerpiece of this Inclusion's design. The results of every gate and module in the trigger funnel down into this wire, and it is used as the input for all gates in the functionality of the Trojan. Therefore, the value iCTRL is a direct representation of whether or not the Trojan has been activated.

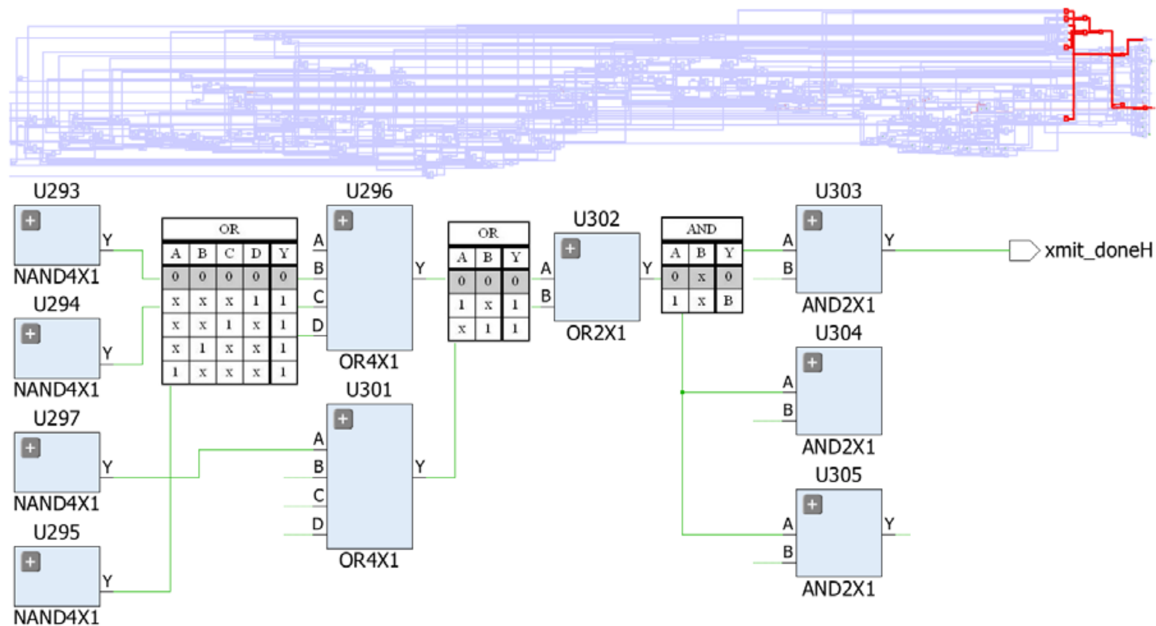


Figure 42. A partial schematic representing a typical layout for the inclusion in the gate-level RS232 circuits. This particular schematic was generated from RS232-T1000. Note that while each of the structures shown here is depicted as a custom module, the labels AND2X1, OR4X1, etc…, reveal them to be implementations of common logic gates. This is a result of uart.v using gates defined in a non-standard Vivado library.

Note that the value representing an activated Trojan is iCTRL = 0_2. This is not listed in the documentation for any of these inclusions. However, every Trojan's functionality is built from one or more AND gates, with iCTRL acting as the A input to each gate. From the AND truth table included in Figure 42, we can see that if iCTRL = 1_2 then the output value of each AND gate will equal to the B input of that gate. In this case, the Trojan is not influencing the circuit outputs, so we can state that it is inactive. A

value of iCTRL = 0_2 forces the output of the AND gate to 0_2, regardless of the value of the B input. The Trojan is actively controlling the values of different wires in the circuit, including, in RS232-T1000, the xmit_doneH output. Therefore, activation occurs when iCTRL = 0_2.

The source values that determine iCTRL vary slightly across benchmarks, but the schematic shown in Figure 42 depicts the most common layout. Specifically, U302's inputs are provided by two OR gates, which accept NAND gate outputs as their own inputs. For the sake of image readability, not all of the NAND gates are shown here. More detailed and complete views will be provided as needed in individual benchmarks.

Using the known value of iCTRL and the schematic shown in Figure 42, we can determine wire values required to activate the Trojan. In particular, using the properties of OR gates, we can establish that the outputs of U296 and U301 must both be 0_2. If either of these wires were to carry a value of 1_2, the iCTRL output would be 1_2, and the Trojan would be inactive. Note that U296 and U301 are also OR gates. Therefore, we can use the same logic to establish that each of their input ports will need a 0_2 value as well. In our discussion of RS232-T1200, we will perform a complete trace to determine all of the source values that affect the wire of iCTRL. This trace should demonstrate the methodology for finding these source values. For all other circuits in this set, we will either provide a list of the triggering values and a short summary of differences from the RS232-T1200 circuit, or we will prove that the Trojan cannot be triggered.

2. RS232-T1000

The first Trojan in the gate-level RS232 group cannot be triggered. This is because the iCTRL = 0_2 condition requires a wire to hold a value of 0_2 and 1_2 at the same time. To prove this, and to provide tools that aid the analysis of later sections, we will continue to trace gate inputs from the inclusion until we can demonstrate this contradictory requirement.

a. Trigger

Figure 43 continues the trace we began in the previous section. We have already established that each input to gate `U296` must be 0_2. From the NAND truth table, this also tells us that every input of gates `U293`, `U294`, and `U295` must be 1_2. Note that `U293` is driven directly by flip-flop outputs. With some exceptions, we can accept flip-flops as value sources in our trace. This is because, using feedback, the flip-flops can retain a value across multiple clock cycles, independent of the value of circuit inputs. For example, the `iXMIT_state_reg` flip-flops form a register that tracks the transmitter's internal state machine. This register changes to track the conversion of a byte of data into a bitstream. However, the value of that byte does not affect the activity of the register. Thus, these flip-flops serve as source values for our trace.

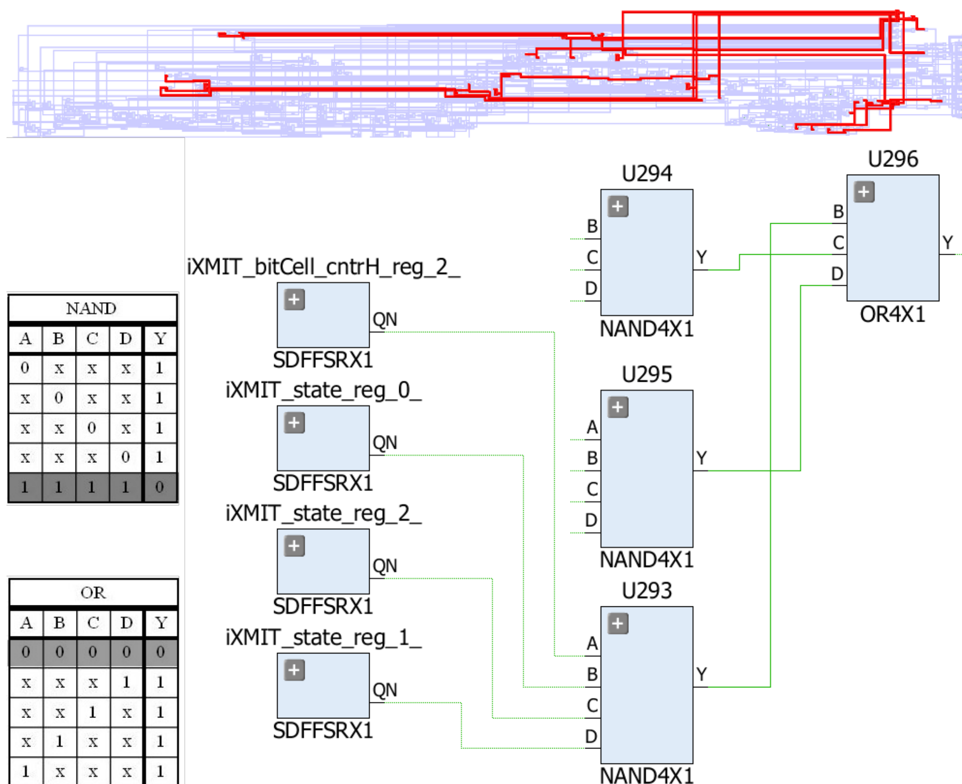


Figure 43. Partial schematic of the inputs to `U296`. Remember that we have previously established that the output and the inputs of `U296` must each have a 0_2 value in order to trigger the Trojan in this circuit. The `NAND` truth table shows the only combination of inputs that will yield 0_2 as an output.

Observe the A input to U293. Flip-flop iXMIT_bitCell_cntrH_reg_2_ provides the value for this input. As stated above, this means that the QN output of iXMIT_bitCell_cntrH_reg_2_ will need to carry a value of 1_2 in order to trigger the Trojan in this circuit.

After using gate U292 to determine the output values of the flip-flops above, we will continue our trace with gate U294. Figure 44 shows this gate, along with the inputs relevant to our discussion. Remember that the output of U294 is 0_2, based on the requirements of U296. The NAND truth table establishes that the inputs of U294 must then be 1_2. U88 and U90, as AND gates, will need 1_2 inputs as well. For U90(A), the input wire is iXMIT_bitCell_cntrH_reg_0_(QN)

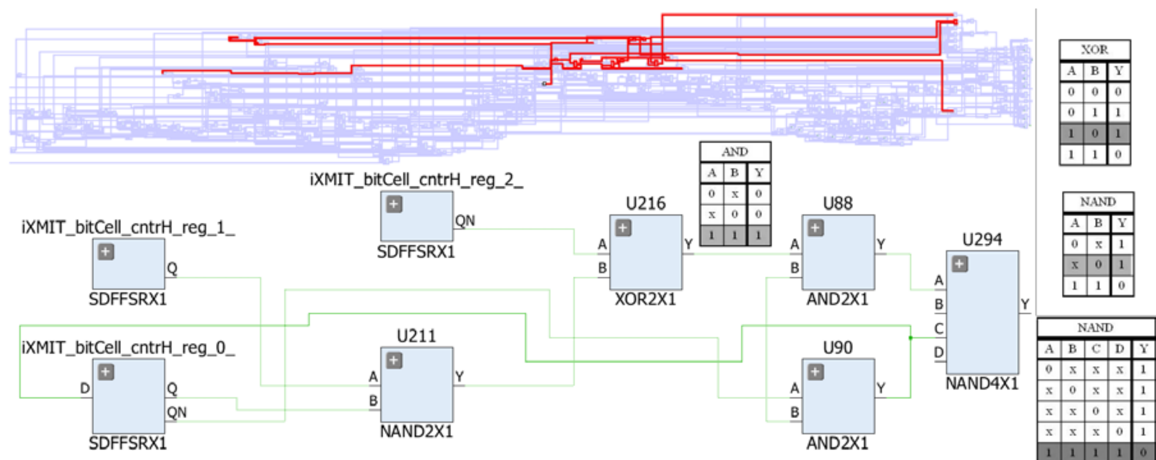


Figure 44. Partial schematic of the inputs to U294. Note that the value of iXMIT_bitCell_cntrH_reg_2_(QN) is being reused. Many of the circuits reuse source values and other gates in the trigger mechanism. This can lead to contradictions like the one discussed here..

Note that iXMIT_bitCell_cntrH_reg_0_ has two output values: Q and QN. Output QN can also be referred to as "Q not," since the design of this flip-flop explicitly establishes these values as opposite. Since QN is 1_2, we know that Q must be 0_2. The Q output of iXMIT_bitCell_cntrH_reg_0_ is used as an input to U211, a NAND gate. As our truth table shows, any 0_2 input forces the output of U211 to 1_2, regardless of other input values.

Now examine U88. Input A, already known to be 1_2, is also the output of U216, an XOR gate. The XOR truth table shows that the gate inputs of U216 must have opposite values. We know the value of the B input from our discussion of U211. Therefore, U216(A) must have a value of 0_2. However, this value is iXMIT_bitCell_cntrH_reg_2_(QN), which we have already established to have a value of 1_2. Based on our analysis, the Trojan can only be activated when iXMIT_bitCell_cntrH_reg_2_(QN) has a value of 0_2 and 1_2 at the same time. Therefore, the Trojan cannot be activated.

b. Functionality

Figure 45 shows the functional portion of this inclusion. Wire iCTRL is used as an input to each of the displayed AND gates. Note that only one of these AND gates directly controls a circuit output. The gate U303 can be used to force xmit_doneH to 0_2, which prevents the RS232 circuit from notifying the larger architecture that it is ready to send a transmission. Gate U305 has some influence over the uart_XMIT_dataH output. In addition to the OAI truth table, we have added an exploded view of the OAI module has been added to Figure 45 to demonstrate the logical design. Note that if input A0 and input A1 are both equal to 0_2, the final module output will be 1_2. Through U305, the malicious inclusion can force A1 to hold a value of 0_2, even when the normal value would be 1_2. If the A0 output is also 0_2, then the output uart_XMIT_dataH will be forced to 1_2. Note that if B0 has a 0_2 value, then the output will already have a 1_2 value, and no impact will come from this portion of the Trojan. Based on the analysis of the OAI module, the U305 portion of the Trojan can be best described as causing reduced accuracy in the transmitted messages.

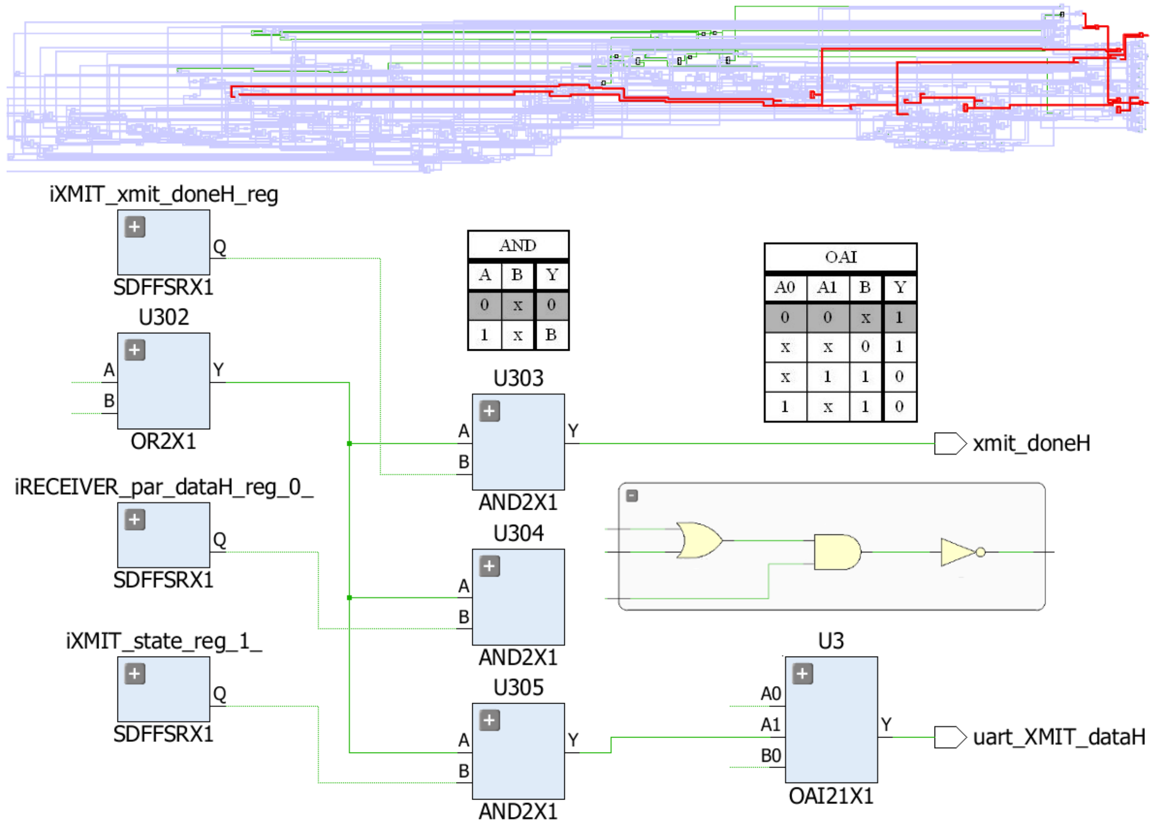


Figure 45. Schematic of the functional portion of `RS232-T1000`. Note that `U303`, `U304` and `U305` are all AND gates. All three of these gates share the common input `iCTRL`. Module `U303` directly controls the circuit output `xmit_doneH`, and `U305` is only separated from `uart_XMIT_dataH` by a single intermediate module. The internal functionality of this OR-AND-invert (OAI) module is shown in the insert above `U3`. Module `U304` does not affect the logical operation of the circuit.

The output of `U304` is listed in the HDL code as `rec_dataH_rec[0]`. Searching the schematics and the HDL code, we have been unable to find another place where this wire is used. We conclude that this portion of the Trojan has no impact on circuit functionality.

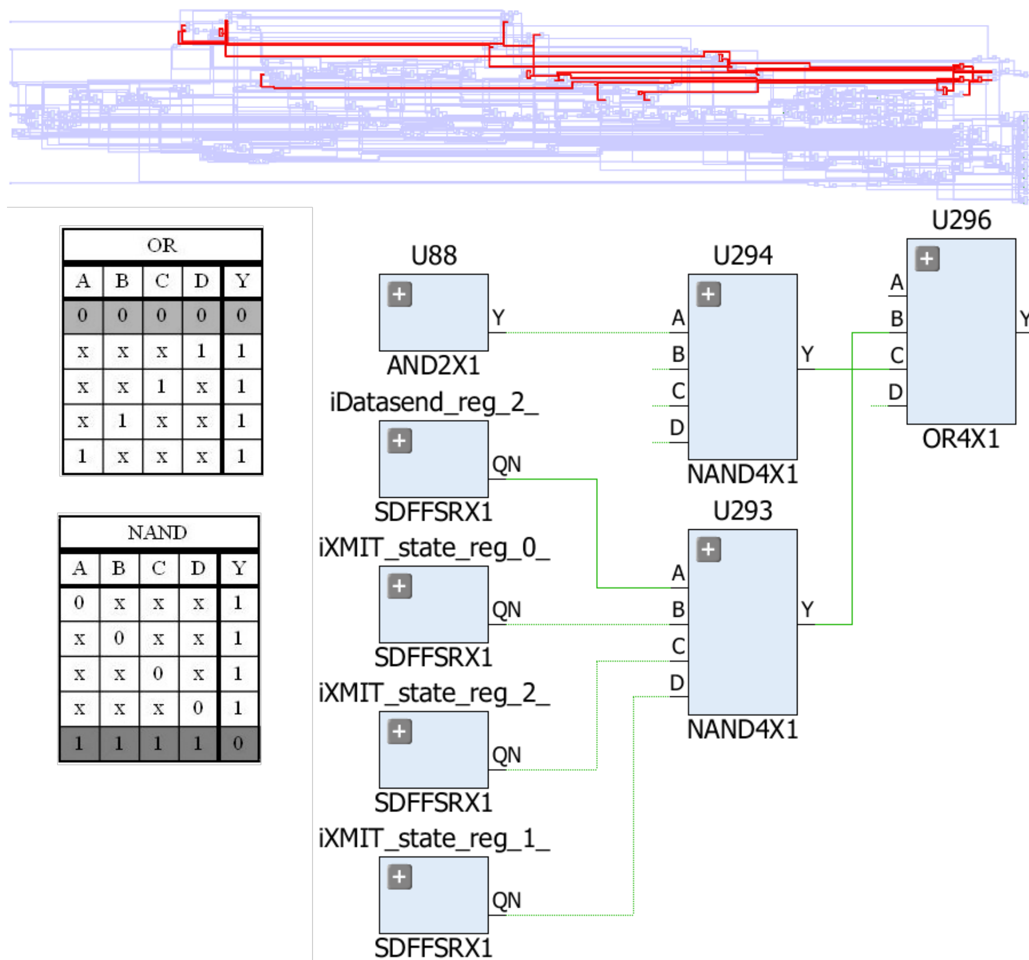
3. RS232-T1100

This Trojan is also unable to trigger due to self-contradictory requirements. Note that as part of the demonstration of this contradiction, we will delve into the internal operation of modules `U91` and `U92`, which are part of the trace of most circuits in this

set. These modules are each constructed of a combination of several different gates, and a detailed knowledge of their structure will add clarity to later discussions.

a. Trigger

Figure 46 demonstrates some of the source inputs to the inclusion in `RS232-T1100`. The `iXMIT_state` flip-flops are particularly relevant to this analysis. Each of these is required to produce a value of 1_2 on their `QN` output in order to meet the requirements imposed by NAND gate `U293`.



OR				
A	B	C	D	Y
0	0	0	0	0
x	x	x	1	1
x	x	1	x	1
x	1	x	x	1
1	x	x	x	1

NAND				
A	B	C	D	Y
0	x	x	x	1
x	0	x	x	1
x	x	0	x	1
x	x	x	0	1
1	1	1	1	0

Figure 46. Partial schematic of `U296` and inputs relevant to this discussion. `U293` and `U294` are both NAND gates with required outputs of 0_2. As a result, all of their inputs must be 1_2. This includes the `QN` outputs from `iXMIT_state`.

Figure 47 shows part of the trace from AND gate U88. By the requirements of U294, we know that U88's output must be 1_2. Only 1_2 inputs will produce this value, so we also know the output of U91. You will note that there are two possible input combinations that can produce the required output of U91. The first combination depends on iXMIT_state_reg_0_(QN). Due to the conditions imposed by U293, we know that this wire must carry a 1_2 input. The A inputs cannot meet the requirements to support U91(Y) = 1_2. Therefore, U91 must receive a 0_2 from its B input, which is provided by U92.

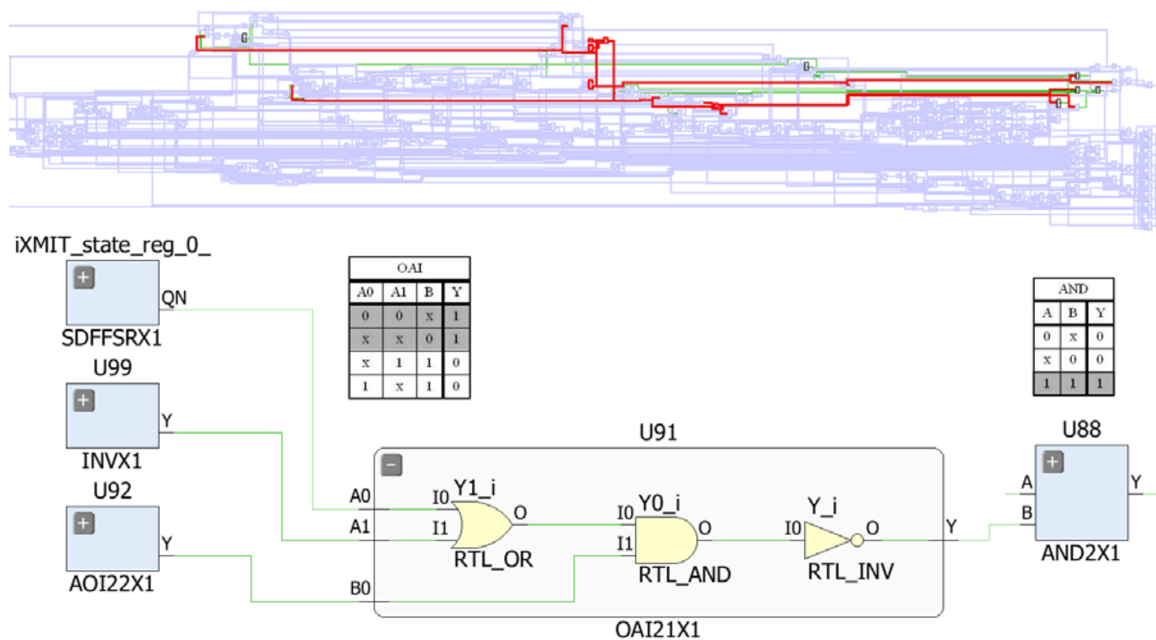
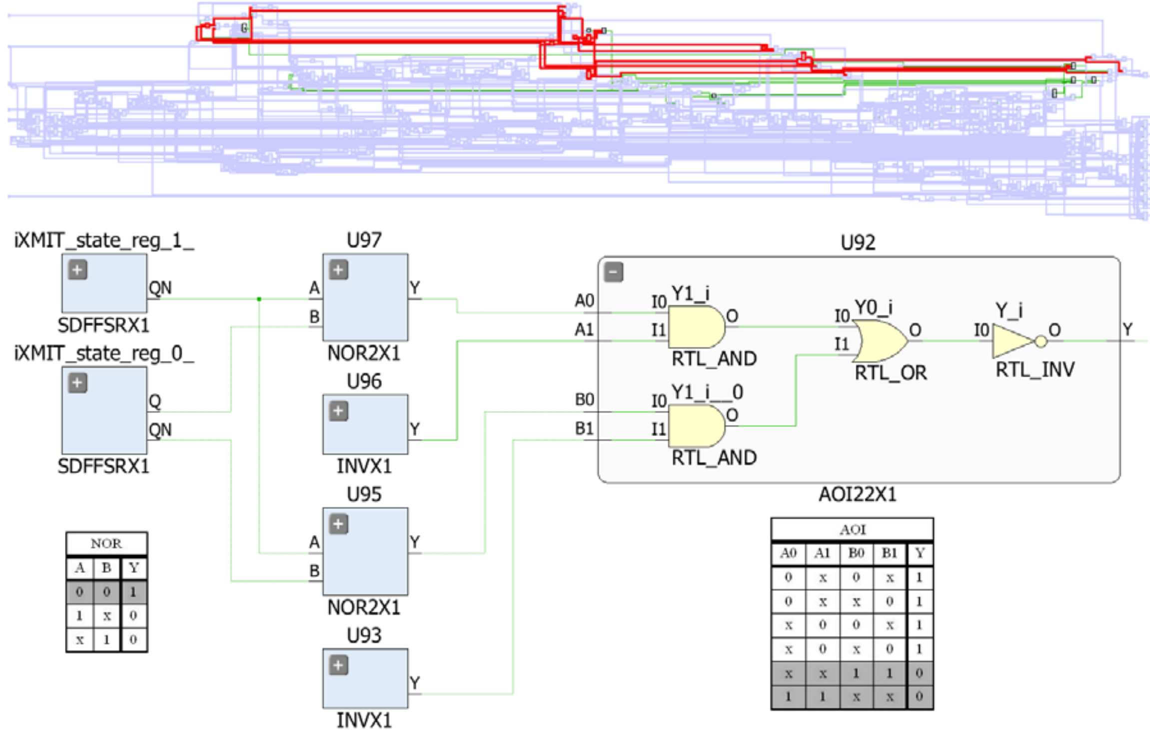


Figure 47. Partial schematic showing a trace from U88. Of particular interest is the module U91. In order to add clarity to later discussions regarding this module, the internal structure is shown here. The truth tables shown here indicate the possible input combinations at each stage of the diagram. Note that only one of the inputs to the AND gate must be 0_2, and that the other can be 1_2 or 0_2 without affecting the final output.

Figure 48 shows the conditions leading to the output of U92. As with U91, this module allows for some flexibility in its inputs. In this case, either U97 and U96 must have 1_2 outputs, or U95 and U93 must have 1_2 outputs.



NOR		
A	B	Y
0	0	1
1	x	0
x	1	0

AOI				
A0	A1	B0	B1	Y
0	x	0	x	1
0	x	x	0	1
x	0	0	x	1
x	0	x	0	1
x	x	1	1	0
1	1	x	x	0

Figure 48. Partial schematic of U292 and its inputs. The AOI truth table describes the possible inputs to this module. Additional tables are used to illustrate the required values at each intermediate stage of the module.

The NOR gate U95 is determined by the QN outputs of iXMIT_state_reg_0_ and iXMIT_state_reg_1_. From U293, we know that both of these values are 1_2. By the NOR truth table, we can show that this input combination leads to an output value of 0_2. The other combination can be eliminated by an examination of U97. This NOR gate also depends on the value of iXMIT_state_reg_0_(QN). Once again, U293 has determined that this value is 1_2. Any 1_2 input to a NOR gate forces the output to 0_2, which prevents U97 from meeting the 1_2 output requirement shown in the AOI truth table. As a result, neither U92 nor U91 can satisfy their output requirements, and the Trojan is impossible to activate.

b. Functionality

The functionality in this benchmark is part of that used in `RS232-T1000`. Figure 49 displays the relevant gates. If it could be triggered, the Trojan would reduce the accuracy of translated messages by occasionally changing a 0_2 bit to a 1_2.

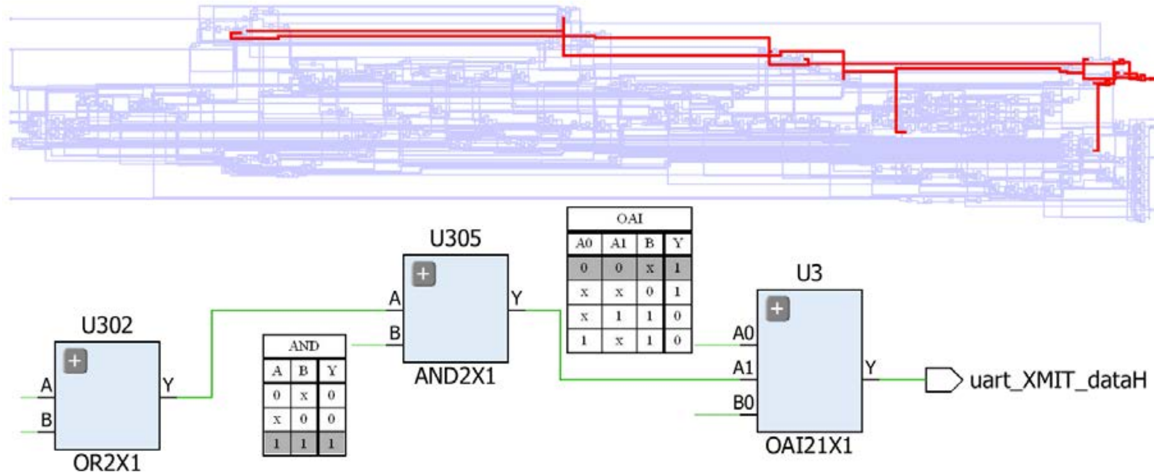


Figure 49. As in `RS232-T1000`, the Trojan functionality is determined by `iCTRL`'s influence over an AND gate. In this case, there is only the single AND gate `U305`, which influences, but does not directly control the transmission output `uart_XMIT_dataH`.

4. RS232-T1200

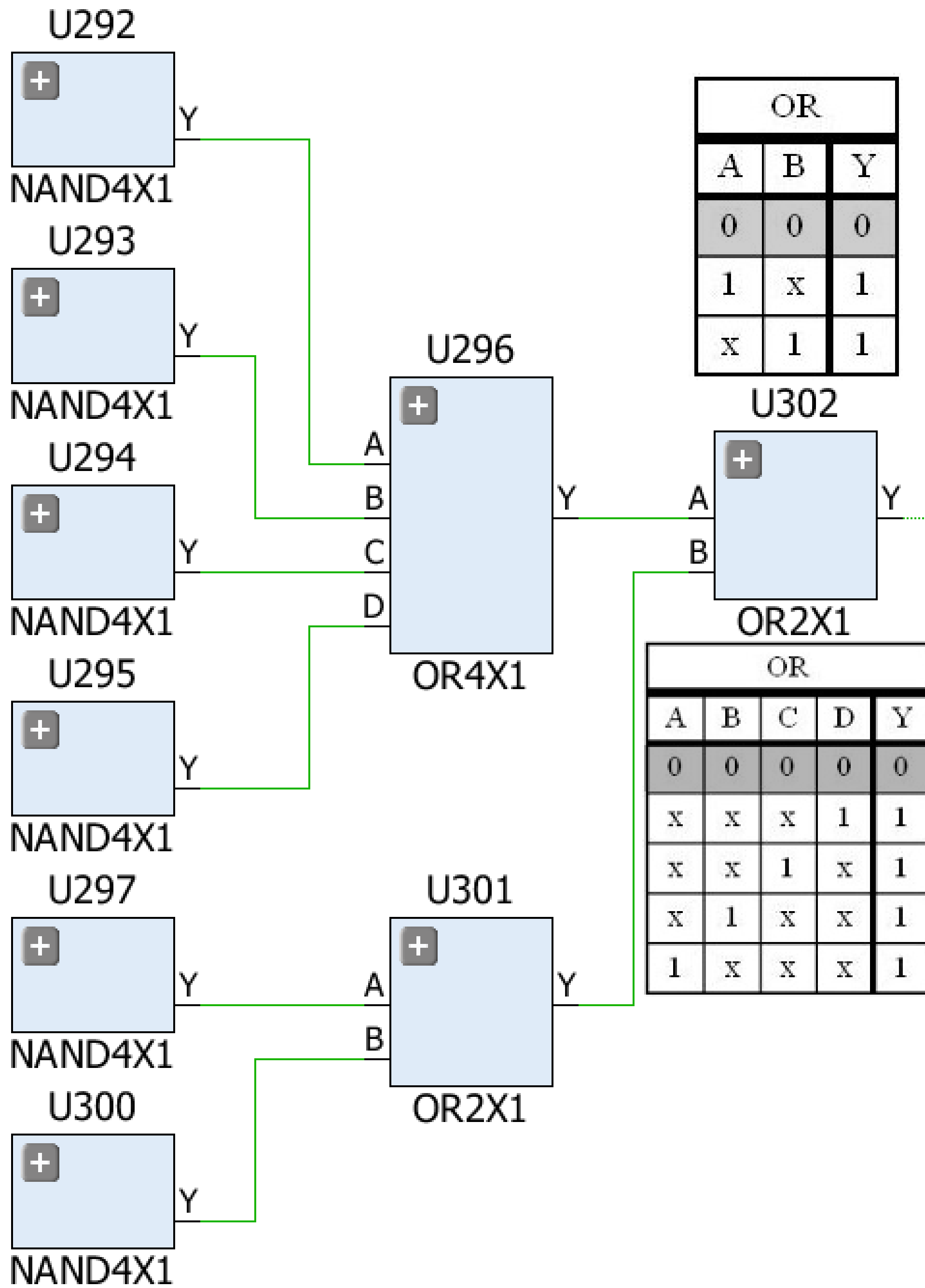
Since `RS232-T1200` is the first Trojan that does not depend on contradictory source values to trigger, we will use it to demonstrate the process of discovering the triggering state required by the gate-level malicious inclusion. In the following section, we will conduct a complete trace of the inputs feeding the Trojan trigger in `RS232-T1200`. This trace will demonstrate the logic used to derive the necessary values and ensure that the circuit does not depend on a self-contradictory state. For the other benchmarks in this set, this thesis will only present the input values and a short summary of the differences between that inclusion and this one.

a. Trigger

We will begin our analysis with a reminder: the Trojan is considered to be active if and only if the wire iCTRL carries a value of 0_2. If iCTRL = 1_2, there is no alteration to the final outputs of the circuit, and the Trojan is inactive. Note that in most of the circuits in this set, iCTRL is the Y output of the OR gate U302. In deference to the HDL code, this thesis will use the name iCTRL.

Figure 50 shows a view of U302 and the two previous levels of the trace for this circuit. Note that iCTRL is the final result of OR gate U302. We know that, while the Trojan is triggered, this output has a value of 0_2. According to the OR truth table, there is only one input combination that will produce this output. Accordingly, we know that U296 and U301 also have outputs of 0_2.

Since U296 and U301 are also OR gates, we can use the same logic to determine that all of their inputs must also be 0_2. This accounts for the outputs of U292, U293, U294, U295, U297, and U300, all of which are 4-way NAND gates. Note that to produce a 0_2 output, each of these gates will need to accept only 1_2 inputs. The truth tables for these gates will be shown as we discuss each of these gates individually.



OR		
A	B	Y
0	0	0
1	x	1
x	1	1

OR				
A	B	C	D	Y
0	0	0	0	0
x	x	x	1	1
x	x	1	x	1
x	1	x	x	1
1	x	x	x	1

Figure 50. Partial schematic showing the trigger of `RS232-T1200`. For space considerations, the inputs of the NAND gates have been omitted, but all 6 are 4-input gates. Note that, to produce an output of `iCTRL` = 0_2, the outputs of each of these NAND gates must be 0_2, as indicated by the OR truth table.

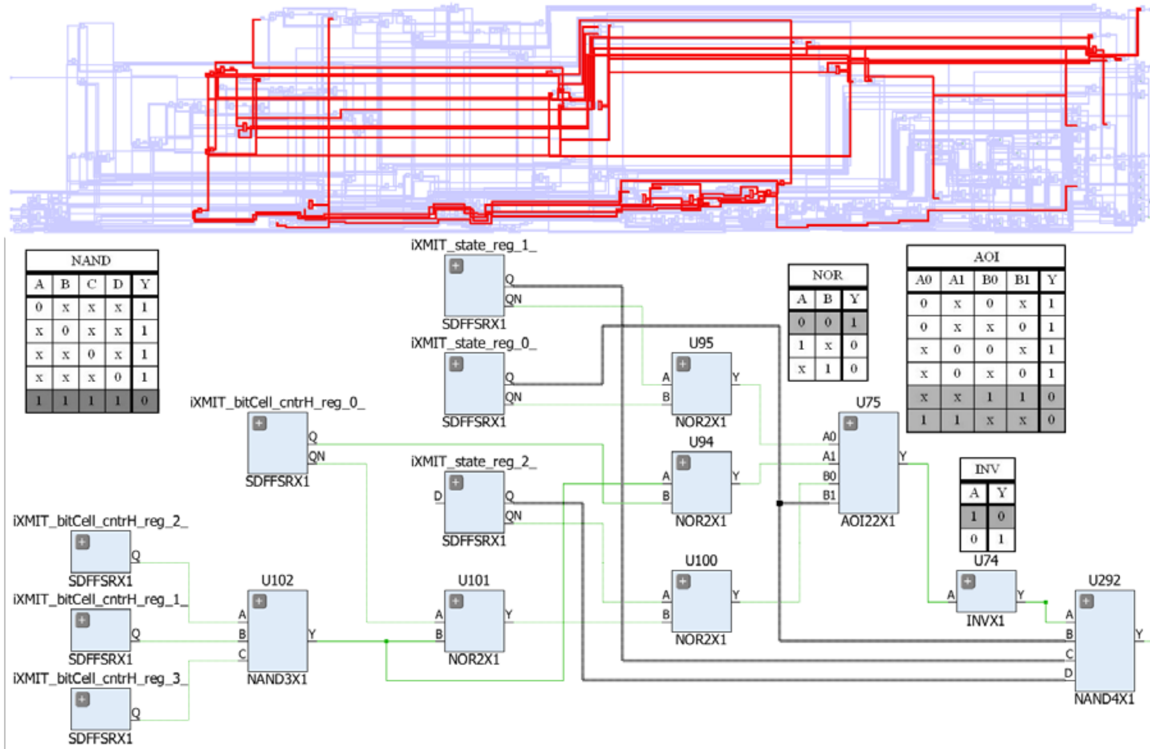
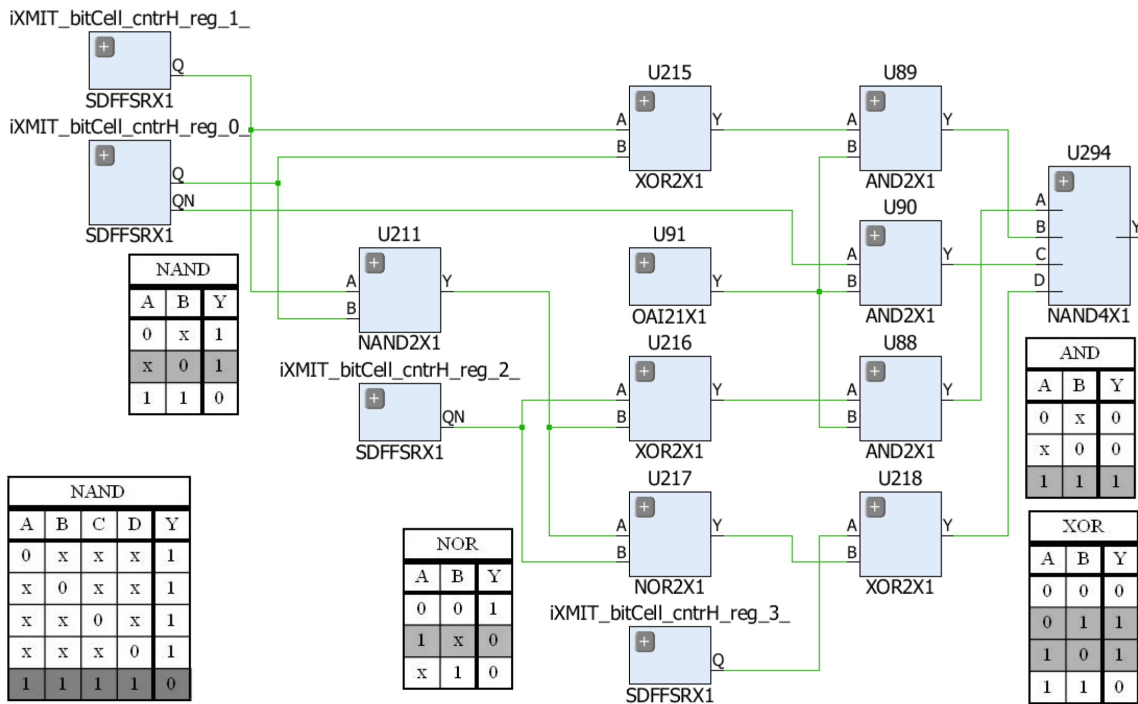


Figure 51. Schematic for the trace of inputs from NAND gate `U292`. This gate directly accepts the `Q` outputs of `iXMIT_state_reg_0_`, `iXMIT_state_reg_1_`, and `iXMIT_state_reg_2_`. We will accept these values as source values.

Figure 51 continues the trace by examining NAND gate `U292`. Note that for the `B`, `C` and `D` inputs, we have already reached flip-flops `iXMIT_state_reg_0_`, `iXMIT_state_reg_1_`, and `iXMIT_state_reg_2_`. We do not need to pursue these inputs further. The `Q` outputs of these gates can each be recorded as requiring a 1_2 value. If we treat `iXMIT_state` as the state register from the transmitter module, we are searching for a transmitter state of 111_2, which is defined in `inc.h` as `x_Datasend`. You may note that this state was not discussed in the RTL section of this thesis. It is not part of the state machine described in the RTL source, and we have been unable to produce this state in simulation of the RS232-T1200 circuit. Note that this is not a contradiction that cannot be resolved, only a challenge that a computer science student was unable to meet.

From U292's A input, our results are less definite. U74, an inverter, must receive a 0_2 input, allowing it to produce a 1_2 value for the NAND gate. The input to U74 comes from the AOI module U75. Observe the AOI truth table. There are actually two sets of inputs that can produce the correct output from this module. For the first set, A0 and A1 must be 1_2. For the second set, B0 and B1 must be 1_2. In either case, we lose the ability to state the value of the other inputs.

Figure 52 shows the trace from U294. As before, all inputs to this gate must be 1_2. For AND gates U88, U89, and U90, this means that they must also have 1_2 inputs. Note that U90(A) is a flip-flop value: iXMIT_bitCell_cntrH_reg_0_(QN). This wire carries a 1_2 value, while the Q output carries a 0_2. This knowledge is instrumental in learning additional source values from this schematic.



NAND		
A	B	Y
0	x	1
x	0	1
1	1	0

NAND				
A	B	C	D	Y
0	x	x	x	1
x	0	x	x	1
x	x	0	x	1
x	x	x	0	1
1	1	1	1	0

NOR		
A	B	Y
0	0	1
1	x	0
x	1	0

AND		
A	B	Y
0	x	0
x	0	0
1	1	1

XOR		
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 52. Schematic of the source values leading to U294. While no flip-flops directly provide inputs to U294, the AND gate U90 still provides a clear value for iXMIT_bitCell_cntrH_reg_0_. Using this value and the properties of U211, U215, U217, and U218, we can determine the requirements for iXMIT_bitCell_cntrH_reg_1_ and iXMIT_bitCell_cntrH_reg_3_.

Examine gate `U215` in Figure 52. We know that the output of this gate is 1_2, because it serves as an input to `U89`. However, `U215` is an XOR gate. A 1_2 output requires that the inputs be opposite: 1_2 and 0_2. The order is not specified by the properties of XOR gates. It is specified, however, by our knowledge of `iXMIT_bitCell_cntrH_reg_0_(Q)`. This wire provides a 0_2 input as `U215`'s `B` value. Now we know that `A`, which is `iXMIT_bitCell_cntrH_reg_1_(Q)`, must be 1_2.

The flip-flops we have discovered allow us to determine a few wire values going forward as well. `U211`, a NAND gate, accepts the exact same inputs as `U215`. In NAND, a (1_2, 0_2) input combination produces a 1_2 output. This output is used as an input to `U217`. We can see from the NOR truth table that this 1_2 input is sufficient to force the output of gate `U217` to 0_2, regardless of the value `iXMIT_bitCell_cntrH_reg_2_(QN)` carries.

There are two more logic gates worth examining in this diagram. First, observe the XOR gate `U216`. We know that, as an input to `U88`, `U216(Y)` must be 1_2. We also have determined that `U211` has a 1_2 output. Since `U216` is an XOR gate, a 1_2 output is the result of accepting a (1_2, 0_2) input. The 1_2 already known, so the other input, `iXMIT_bitCell_cntrH_reg_2_(QN)`, must be 0_2.

Now we can examine `U218`. We know that the output of this gate must be 1_2, to satisfy `U294`. We also know that the `B` input of this gate is 0_2. These two pieces of information allow us to determine that the `A` input, `iXMIT_bitCell_cntrH_reg_3_(Q)`, must be 1_2. Thus, from the schematic above, we have discovered the following properties of an activated Trojan circuit:

`iXMIT_bitCell_cntrH_reg_0_(Q)` = 0_2

`iXMIT_bitCell_cntrH_reg_1_(Q)` = 1_2

`iXMIT_bitCell_cntrH_reg_2_(Q)` = 1_2

`iXMIT_bitCell_cntrH_reg_3_(Q)` = 1_2.

Note that when discussing flip-flops, we will attempt to maintain consistency by identifying the Q output of each module, even if the QN output was the first output discovered. This also allows us to discuss the equivalent register value of a group of flip-flops. For example, the values listed here correspond to a register value of 0xE. In the RTL version of this circuit, bitCell_cntrH = 0xE was the signal for the transmitter state machine to place the next sequence bit on wire uart_xmit_dataH. This value is reproduced on a regular basis as part of the circuit's operation.

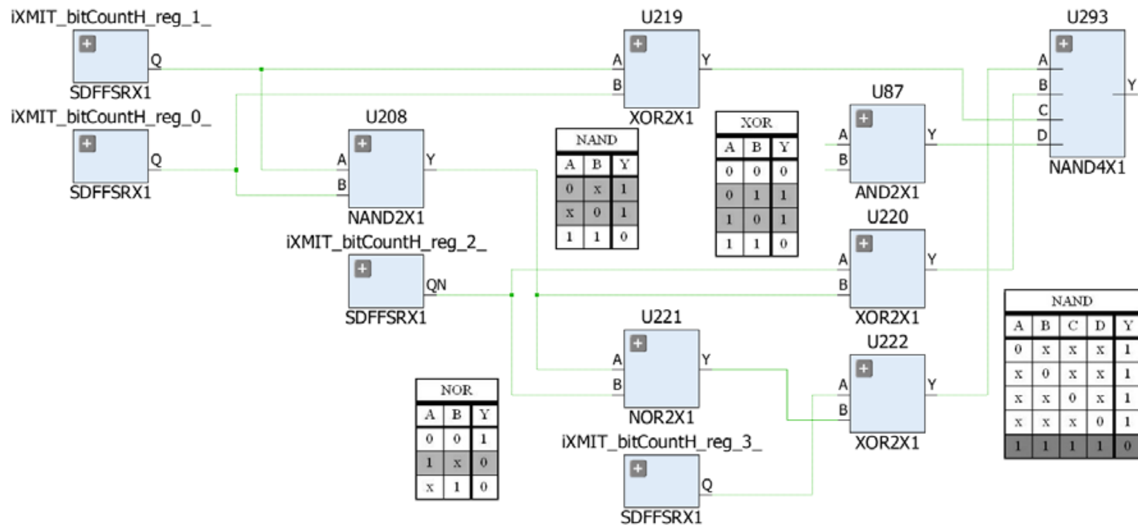


Figure 53. Schematic of the source values leading to U293. The flip-flops shown here correspond to the RTL register bitCountH. Note that from this schematic, we can determine a relationship between iXMIT_bitCountH_reg_1_(Q) and iXMIT_bitCountH_reg_0_(Q), but we cannot assign precise values to them.

We will continue our analysis using Figure 53. The displayed module U293 is one of the NAND gates that provides a 0_2 value for U296. Thus, we know that all inputs to U293 must be 1_2. For U219, an XOR gate, this means that iXMIT_bitCountH_reg_1_(Q) and iXMIT_bitCountH_reg_0_(Q) must form a (1_2, 0_2) pair, but it does not determine the ordering of that pair. The only rule imposed on these values is that they must be opposite. Note that this knowledge is still

useful. These two values are reused as inputs to U208, a NAND gate. Since the inputs of this gate include at least one 0_2, the output Y must be 1_2.

Knowing the output of U208, we will now examine U220. By our analysis of U293, we know that U220(Y) is 1_2. Input U220(B) is the 1_2 from U208. With these two pieces of information, we can determine that the value of U220(A), iXMIT_bitCountH_reg_2_(QN), is 0_2.

The last flip-flop value we can determine from this schematic is iXMIT_bitCountH_reg_3_(Q), which is used as the A input of U222. Like U220, U222 is an XOR gate with an output of 1_2. The B input of U222 is U221(Y). Note that U221 is a NOR gate that accepts U208(Y) and iXMIT_bitCountH_reg_2_(QN), for a (1_2, 0_2) input combination. That means U221(Y) must be 0_2. Since U222 needs opposite inputs to produce a 1_2 output, iXMIT_bitCountH_reg_3_(Q) must be 1_2.

After examining the schematic in Figure 53, we can state the following:

iXMIT_bitCountH_reg_3_(Q) = 1_2

iXMIT_bitCountH_reg_2_(Q) = 1_2

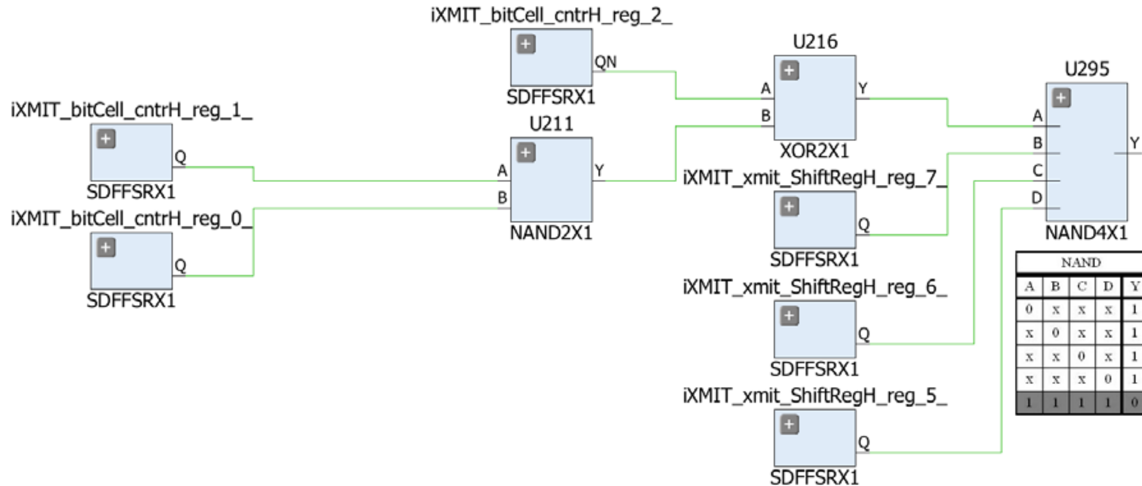
iXMIT_bitCountH_reg_1_(Q) != iXMIT_bitCountH_reg_0_(Q)

Due to the flexibility resulting from iXMIT_bitCountH_reg_1_(Q) and iXMIT_bitCountH_reg_0_(Q), these results show that the trigger can be activated if the register bitCountH has a value of either 13_{10} or 14_{10}. Register bitCountH is meant to count bits of one message as they are transmitted. After a complete mesage has been transmitted, the register will be reset to 0_{10} and the count will begin again.

Note that a value of 13_{10} or 14_{10} on bitCountH may be unachievable. The RS232 circuit is designed to transmit and receive single bytes. Under this design, bitCountH should never hold a value greater than 8_{10}. In the gate-level design, input bus xmit_dataH and output bus rec_dataH are both 8 bits wide. If the circuit counts a message longer than 8 bits, it has no means to complete the conversion. Note that we are discussing the value of bitCountH in base 10 notation because that is the

notation used to assign the value of this register in the HDL code defining the RTL version of the circuit.

Figure 54 shows the source values for NAND gate U295. The A value is determined by U216, which we have already examined as part of our study of U294. Recall that, as an input to U88, U216 was required to have a value of 1_2 on its Y output. We can confirm that, because U216(Y) is the A input to U295, this requirement holds without causing a contradiction. Comparing the requirements for each reuse of a module allows us to confirm that the activation of RS232-T1200 does not depend on contradictory source values.



A	B	C	D	Y
0	x	x	x	1
x	0	x	x	1
x	x	0	x	1
x	x	x	0	1
1	1	1	1	0

Figure 54. Schematic of the source values leading to U295. Note that the A input of U295 is provided by U216(Y), which we discussed in our examination of U294. This reuse of source gates is common among the gate-level RS232 circuits.

Module U295 also tells us the values of three new flip-flops. iXMIT_xmit_ShiftRegH_reg_7_(Q), iXMIT_xmit_ShiftRegH_reg_6_(Q), and iXMIT_xmit_ShiftRegH_reg_5_(Q) must each have a value of 1_2. Note that these flip-flops represent the high-order bits of the register xmit_ShiftRegH, which is used as an intermediate storage for the byte currently being transmitted. Based on the

RTL version of the circuit, bit [0] of the register is used as the value of uart_XMIT_dataH. After the appropriate time has elapsed, all bits in the register are shifted one position. The previous bit [1] is now bit [0] and is being transmitted on uart_XMIT_dataH. The empty value of bit [7] is filled with a 1_2. After another bit is transmitted, this 1_2 will be shifted to bit [6]. Note that after 3 bits of a message have been transmitted, the Q outputs of these flip-flops are all guaranteed to carry the value 1_2. Therefore, this Trojan activation condition can be met regardless of the value of circuit input bus xmit_dataH.

You may notice that Figures 52 and 53 include gates that have not yet been discussed. This is a deliberate choice, because gates U91 and U87 are both linked to source values we have already examined. The analysis so far has been focused on finding the source values for the trigger in RS232-T1200 by the most efficient means possible. Since U91 and U87 have more ambiguity in their inputs, we have chosen to use them for the purpose of confirming the lack of contradictory requirements in this Trojan's trigger input sources.

We will begin by examining U91, which is shown in Figure 55. We know, from U88, U89, and U90 that U91 needs a 1_2 output. In the interest of verification, we will actually approach this problem from our known flip-flop values. If we can establish a 1_2 value for U91(Y), then we can state that our required source values are free of contradiction.

The first input to U91 is the Y output of U102, which is a NAND gate accepting iXMIT_bitCell_cntrH_2_(Q), iXMIT_bitCell_cntrH_1_(Q), and iXMIT_bitCell_cntrH_3_(Q) as inputs. Since all three of these inputs are 1_2, we know that U102(Y) = 0_2. This value, combined with a 1_2 output from iXMIT_bitCell_cntrH_0_(QN), results in U101(Y) = 0_2. We will use this and iXMIT_state_reg_2_(QN) = 0_2 to establish that U100(Y) = 1_2. Through inverter U99, the A1 input of U91 becomes 0_2. Value iXMIT_state_reg_0_(QN)

is also 0_2, meaning that both items in the A pair of U91 are 0_2. By the AOI truth table shown in Figure 55, U91(Y) has been confirmed as 1_2.

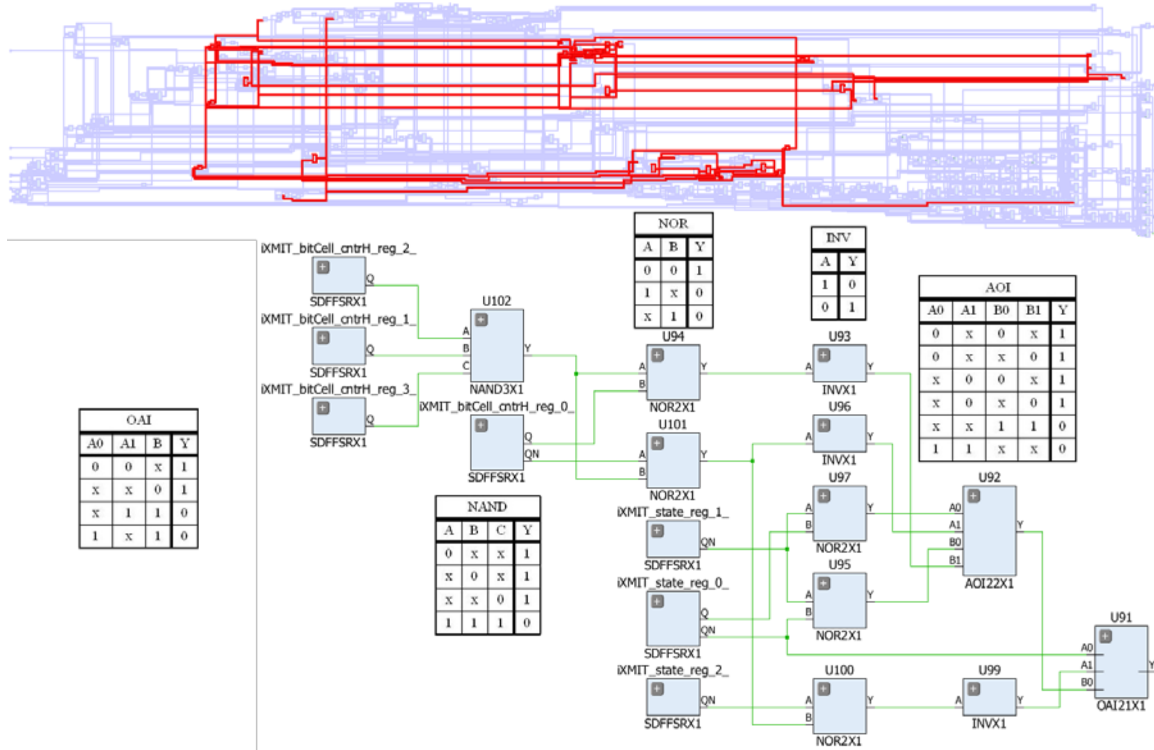


Figure 55. Partial Schematic of the source values for U91. Note that the values for every flip-flop shown here have already been determined. We will use this diagram for verification purposes.

Observe Figure 56. The final gate in this structure is U87, which connects directly to U293, as shown in Figure 53. This means that U87's final output must be 1_2. Note that U87 accepts inputs from U218 and U91, both of which have been used in previous schematics. We have proven that, based on our known inputs, U91 has a 1_2 input. The XOR gate U218 was instrumental in determining the value of iXMIT_bitCell_cntrH_3_(Q). From U294, we know that U218(Y) carries a 1_2 input. Since we have previously examined the source values of both of these gates, we can accept these 1_2 values as valid and prove that U87(Y) is indeed 1_2.

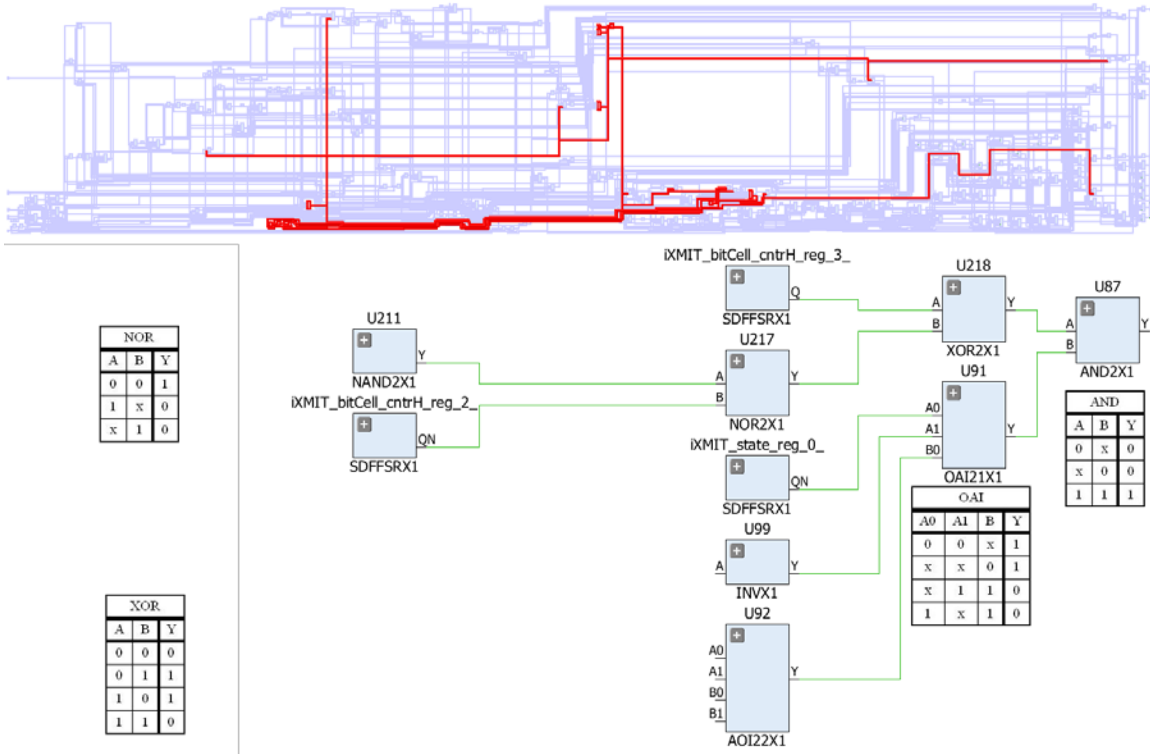


Figure 56. Partial schematic of the inputs of `U87`. As with `U91`, we will use known values to demonstrate that `RS232-T1200` does not contain a contradiction.

With this confirmation, we have completed the trace of the source values to `U296`. Note that all of the flip-flop sources we discovered were part of the transmitter portion of the `RS232` circuit. This seems to be a deliberate design choice by the inclusion's designers. The second portion of our analysis will focus on the gates that serve as inputs to `U301`. The source values for this portion of the trigger are drawn from the receiver portion of the circuit. We will begin by discussing `U297`, which is determined by the values of 2 registers.

Figure 57 shows the flip-flop sources that are used to determine the output of `U297`. Recall that we have already established that output `U297(Y)` must carry a value of 0_2. Because `U297` is a NAND gate, all four of its inputs must be 1_2, according to the same logic used to determine input values for `U292`, `U293`, `U294`, and `U295`. This means we have already determined that `iRECEIVER_state_reg_2_(Q) =` 1_2,

iRECEIVER_state_reg_1_(Q) = 1_2, and iRECEIVER_state_reg_0_(Q) = 1_2.

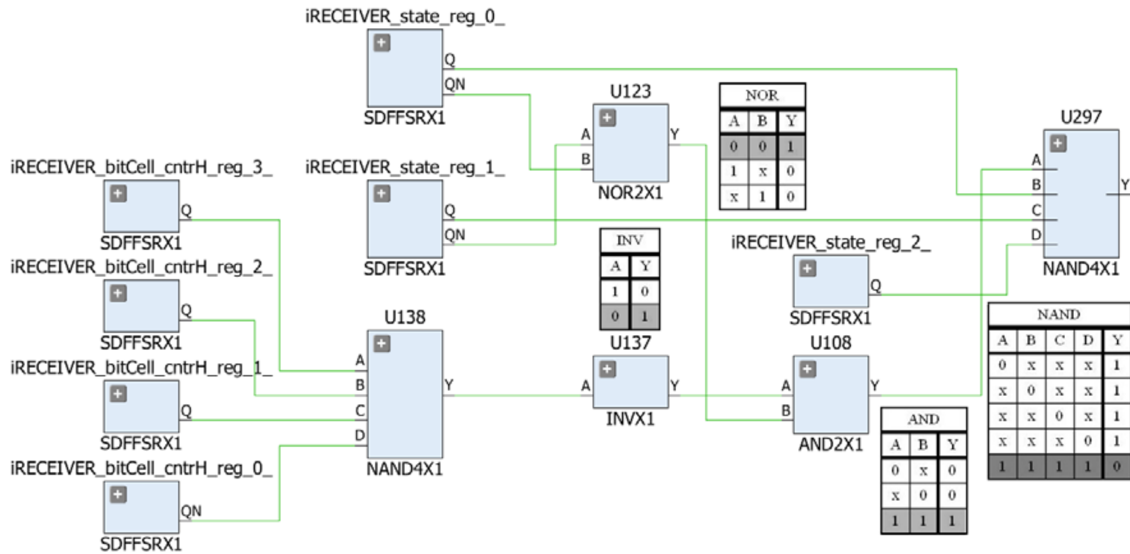


Figure 57. Schematic of the source values leading to U297. Note that U297 can be used to directly determine the iRECEVER_state values, while iRECEIVER_bitCell_cntrH requires several stages of analysis. However, there are no ambiguous input combinations in this set. Each gate's required output allows for only one possible combination of inputs.

We will now investigate U297's A input, which is provided by the Y output of U108. If this wire is 1_2, then U108(A), which is U137(Y), and U108(B), which is U123(Y), must both be 1_2 as well.

U123 is a NOR gate, which can only produce a 1_2 output if both inputs are 0_2. These 0_2s are the QN outputs of iRECEIVER_state_reg_0_ and iRECEIVER_state_reg_1_. Note that this information is simply a confirmation of the previously discovered values iRECEIVER_state_reg_1_(Q) = 1_2, and iRECEIVER_state_reg_0_(Q) = 1_2.

Gate U108's other input is the result of inverter U137. The inverter acts as a simple NOT gate, so U137's input, U138(Y), must be 0_2. We have already discussed

NAND gates like U138. If the output is 0_2, the inputs must all be 1_2. This gives us the values for all of the iRECEIVER_bitCell_cntrH flip-flops. For iRECEIVER_bitCell_cntrH_0_, U138 uses the QN output, so the Q output is actually 0_2. For the other three flip-flops, U138 uses the Q output as its source value. This means that each of those Q outputs has a value of 1_2.

In summary, we have discovered the following source values from U297:

iRECEIVER_state_reg_2_(Q) = 1_2

iRECEIVER_state_reg_1_(Q) = 1_2

iRECEIVER_state_reg_0_(Q) = 1_2

iRECEIVER_bitCell_cntrH_3_(Q) = 1_2

iRECEIVER_bitCell_cntrH_2_(Q) = 1_2

iRECEIVER_bitCell_cntrH_1_(Q) = 1_2

iRECEIVER_bitCell_cntrH_0_(Q) = 0_2

These values represent a state of 111_2 for the receiver state machine and a bitCell_cntrH value of 0xE. Note that this state is not defined in the RTL file inc.h, and we have been unable to produce it in simulation. However, bitCell_cntrH = 0xE is a timing condition that is guaranteed to occur once for every bit that is received by this circuit.

The NAND gate U300 is shown in Figure 58. Trojan activation requires that U300(Y) = 0_2, and thus all inputs to U300 must be 1_2. This proves simple to analyze as the inputs to U300 are directly provided by the iDatasend flip-flops shown in Figure 58. These flip-flops are documented elements of the inclusion, and we will demonstrate the analysis of iDatasend_reg_1 in order to provide guidance for later analysis.

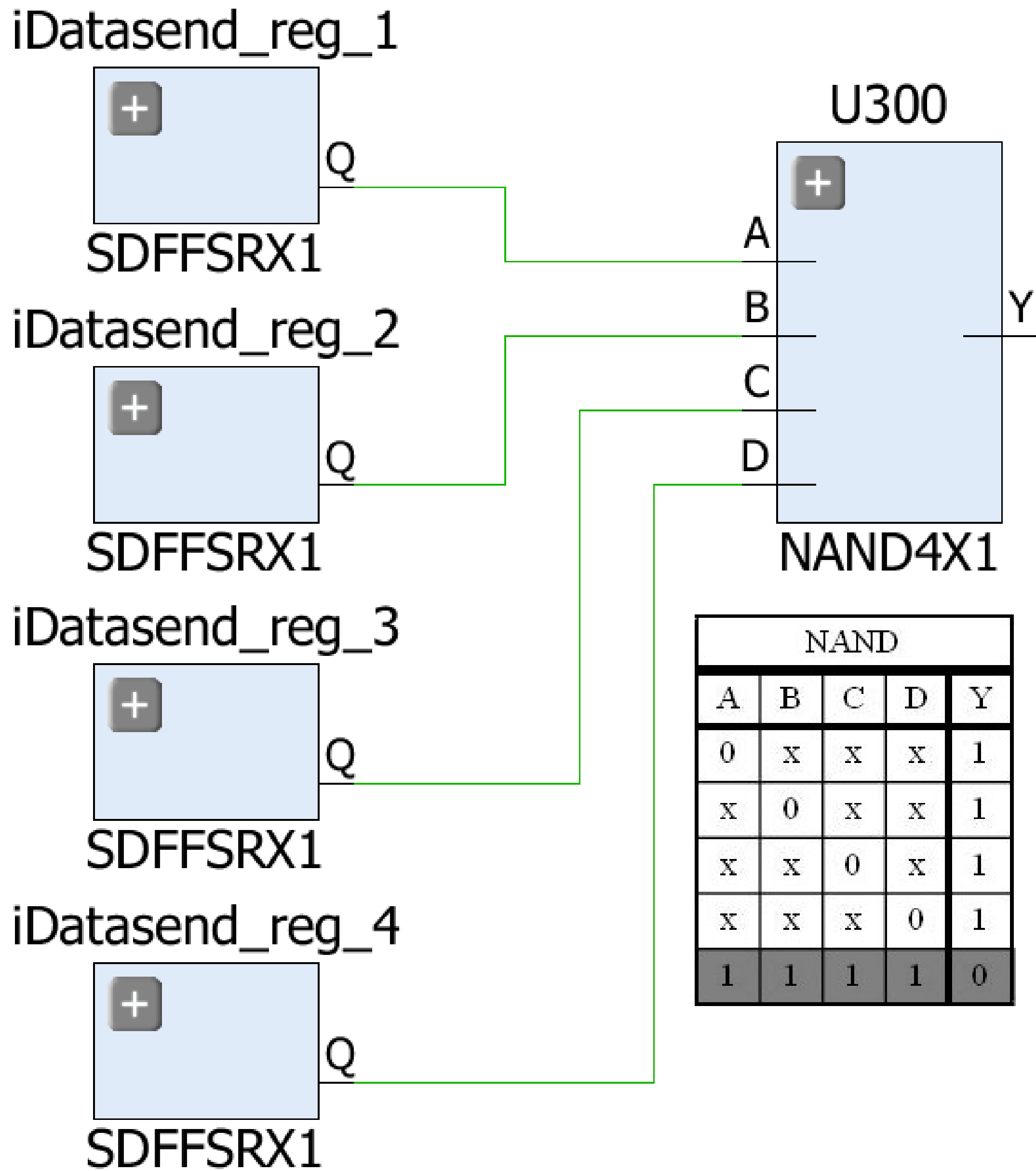


Figure 58. Schematic of the source values leading to U300. Note that all of these values are Q outputs from inserted flip-flop modules.

Figure 59 displays the source values of the iDatasend_reg_1 flip-flop. Because this flip-flop was added to the as part of the malicious inclusion, we will analyze its inputs more thoroughly. First, observe input SN. Vivado's schematics use this wire shape to represent a constant 1_2. For our purposes, this means that the condition SN = 0, which is part of the bottom two rows of the SDFFSR truth table, can never exist for

this flip-flop. There are now only two possible combinations that can produce a `Q = 1` output from this flip-flop.

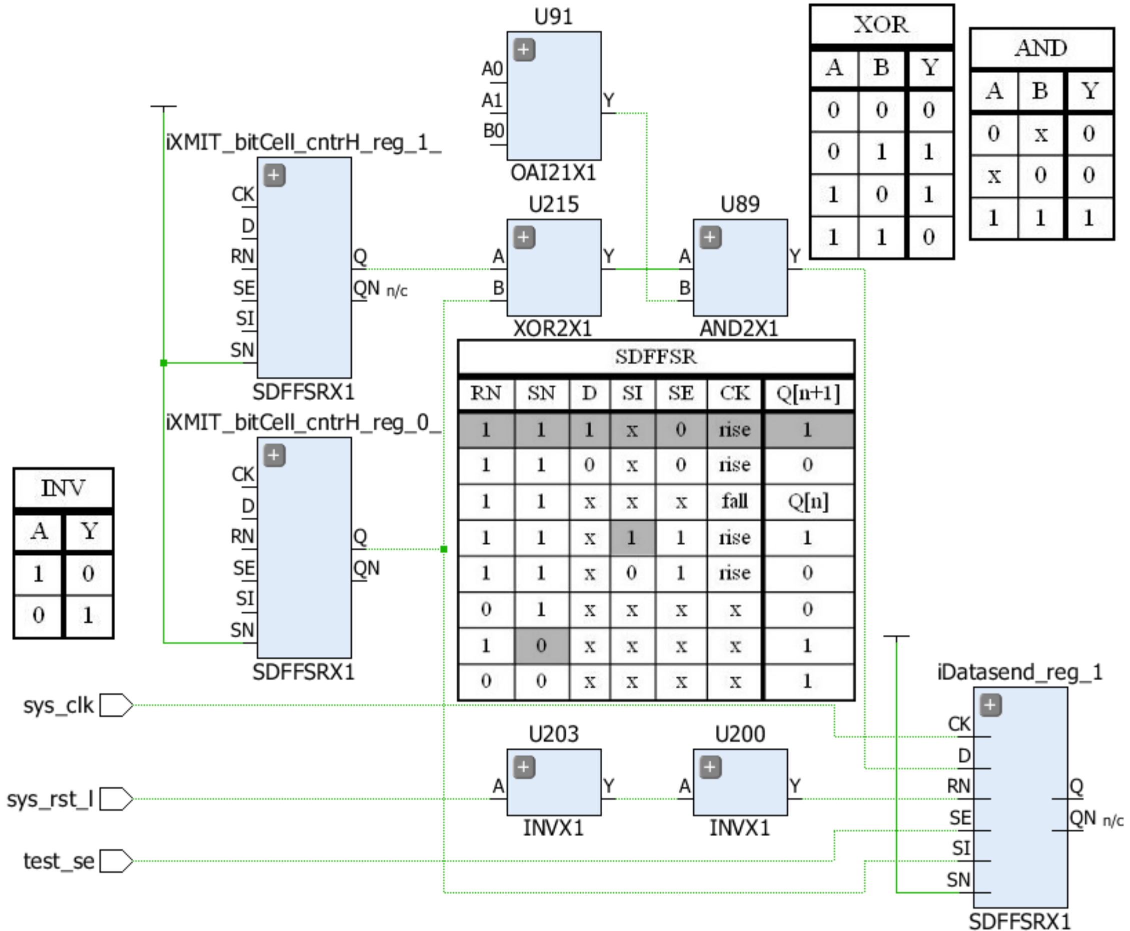


Figure 59. Schematic of the source values leading to `iDataSend_reg_1`. Note that `SN` is set to the constant 1_2, meaning that input combinations in the bottom most rows of the SDFFSR truth table will never be observed.

We can eliminate one of these combinations from consideration by examining the `SI` input. This input is drawn from `iXMIT_bitCell_cntrH_0_(Q)`. In row 4 of the `SDFFSR` truth table, we can see a combination that requires `SI` to be 1_2. We have already established that, during Trojan operation, `iXMIT_bitCell_cntrH_0_(Q)` $= 0_2$. Changing this value alone will cause `iDatasend_reg_1(Q)` to be set to 0_2 at the start of the next clock cycle. This action would deactivate the Trojan.

The first row of the SDFFSR truth table is the only one that can produce a sustained 1_2 value. This can be accomplished by four set values being maintained on the inputs to this flip-flop.

First, RN is required to be 1_2. RN is produced by a double inversion of sys_rst_l, which means that RN = sys_rst_l. Recall from our discussion of circuit features that sys_rst_l is required to maintain a 1_2 input during normal circuit operation. The SN = 1 condition is also easily met because the input is the constant 1_2. The value of input SE is directly provided by circuit input test_se. The user can directly set this value to 0_2. Finally, the D = 1_2 flip-flop requirement is met because it is provided by U89(Y), which is also an input to U294. Therefore, if all other triggering conditions are met, iDatasend_reg_1(Q) = 1 adds only the condition test_se = 1_2.

Similar examinations of iDatasend_reg_2, iDatasend_reg_3, and iDatasend_reg_4 will reveal two additional requirements: input UART_rec_dataH = 1_2 and iRECEIVER_rec_datSyncH_reg(Q) = 1_2. These are the last driving values for this particular trigger mechanism.

Table 1 presents a summary of the register values we discovered during the analysis of this circuit. Each set of related flip-flops is grouped together so that the associated register value can be displayed in association with the appropriate values.

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iRECEIVER_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iXMIT_state_reg_#_		1	1	1	111_2
iRECEIVER_state_reg_#_		1	1	1	111_2

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 1. Table of register values for RS232-T1200. This table represents grouped flip-flop Q values required to activate the Trojan in RS232-T1200. We have presented the flip-flops in this fashion so that we can also present the value of the RTL register, which is useful in explaining the purpose of each flip-flop group.

Note that in addition to the values shown in Table 1, RS232-T1200 requires the following values to activate the inserted Trojan:

- UART_rec_dataH = 1_2
- sys_rst_l = 1_2
- test_se = 1_2
- iRECEIVER_rec_datSyncH_reg(Q) = 1_2

If all of the above conditions are met, iCTRL will carry a 0_2 value, and the Trojan will be active.

b. Functionality

The Trojan in this circuit accomplishes part of the functionality shown in RS232-T1000. In Figure 60, you will observe that iCTRL can be used to force the output xmit_doneH to 0_2. This prevents the RS232 circuit from notifying the larger architecture that the most recent message has been sent. Without this notification, the architecture will not submit another message, and the system will suffer a denial-of-service.

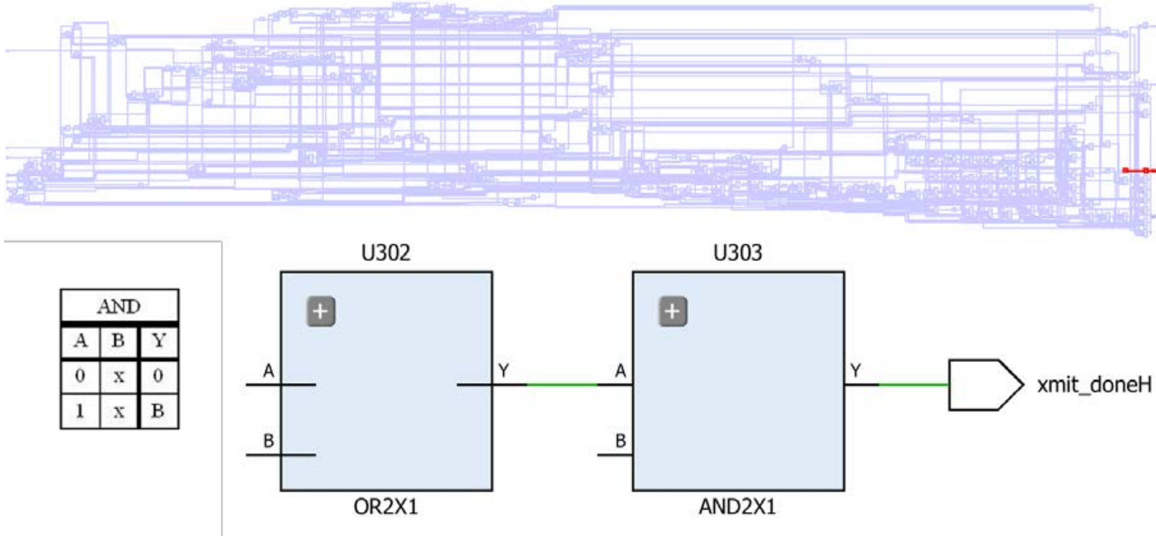


Figure 60. The Trojan functionality of RS232-T1200. Note that the value of gate U303 is used as the output xmit_doneH. This allows the iCTRL wire to force that output to 0_2 after the Trojan is triggered.

5. RS232-T1300

a. Trigger

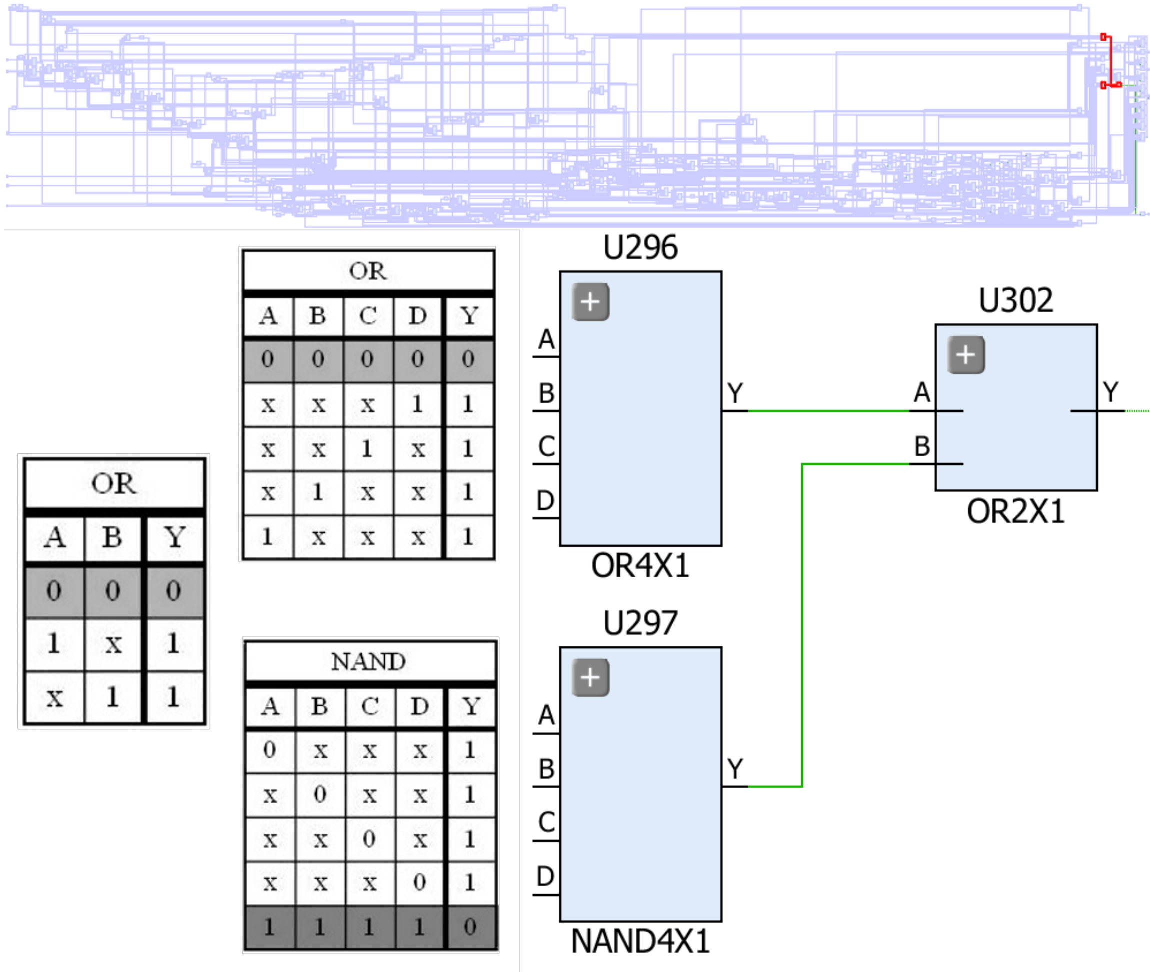
The triggering source values of RS232-T1300 are shown in Table 2. If the Q output of these flip-flops carries the value shown, then iCTRL will carry a 0_2 value, and the Trojan will be activated. Note that the "value" column was added based on the apparent relationship between these flip-flops and the registers found in the RTL version of RS232. We selected the base of each value according to the HDL from RS232-T100. For example, the state values are defined in file inc.h as 3-digit binary values, while WORD_LEN, which is used in conjunction with register iXMIT_bitCount, is defined as 8_{10}.

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iRECEIVER_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iXMIT_state_reg_#_	■	1	1	1	111_2
iRECEIVER_state_reg_#_	■	1	1	1	111_2

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 2. Table of source values for the RS232-T1300 trigger mechanism. The flip-flops have been grouped into logical registers, to demonstrate the full value of each group. Note that iXMIT_xmit_ShiftRegH_reg is actually an 8-bit register, but that bits 0 through 4 have no impact of the Trojan trigger. Also note that the outputs iXMIT_bitCount_reg_1_(Q) and iXMIT_bitCount_reg_0_(Q) must have opposite values, but that the order of those values does not alter the result of the Trojan trigger.

The Trigger of RS232-T1300 is similar to that of RS232-T1200, but it is missing several elements, as shown in Figure 61. In particular, most of the receiver side logic has been removed from the trigger mechanism. U302 draws its B input directly from NAND gate U297. Note that U297 still needs to provide an output of 0_2 to trigger the Trojan. Source value requirements for this gate also remain unchanged from those in RS232-T1200. However, U298, U299, U300 and their source values are no longer relevant to the operation of this Trojan. In particular, the iDatasend flip-flops have been completely removed from this circuit.



OR		
A	B	Y
0	0	0
1	x	1
x	1	1

OR				
A	B	C	D	Y
0	0	0	0	0
x	x	x	1	1
x	x	1	x	1
x	1	x	x	1
1	x	x	x	1

NAND				
A	B	C	D	Y
0	x	x	x	1
x	0	x	x	1
x	x	0	x	1
x	x	x	0	1
1	1	1	1	0

Figure 61. Partial schematic of the trigger for RS232-T1300. This Trojan has a slightly different structure than that used in other benchmarks in this group. The OR gate U301 has been removed from the structure, and NAND U297 is directly connected to U302.

b. *Functionality*

RS232-T1300 uses similar functionality to the previous gate-level Trojans. When triggered, this Trojan will force the outputs rec_readyH and xmit_doneH to 0$_2$, as shown in Figure 62. This results in a denial-of-service attack against the system. While these signals are 0$_2$, the architecture will not collect data from the receiver or attempt to send it through the transmitter.

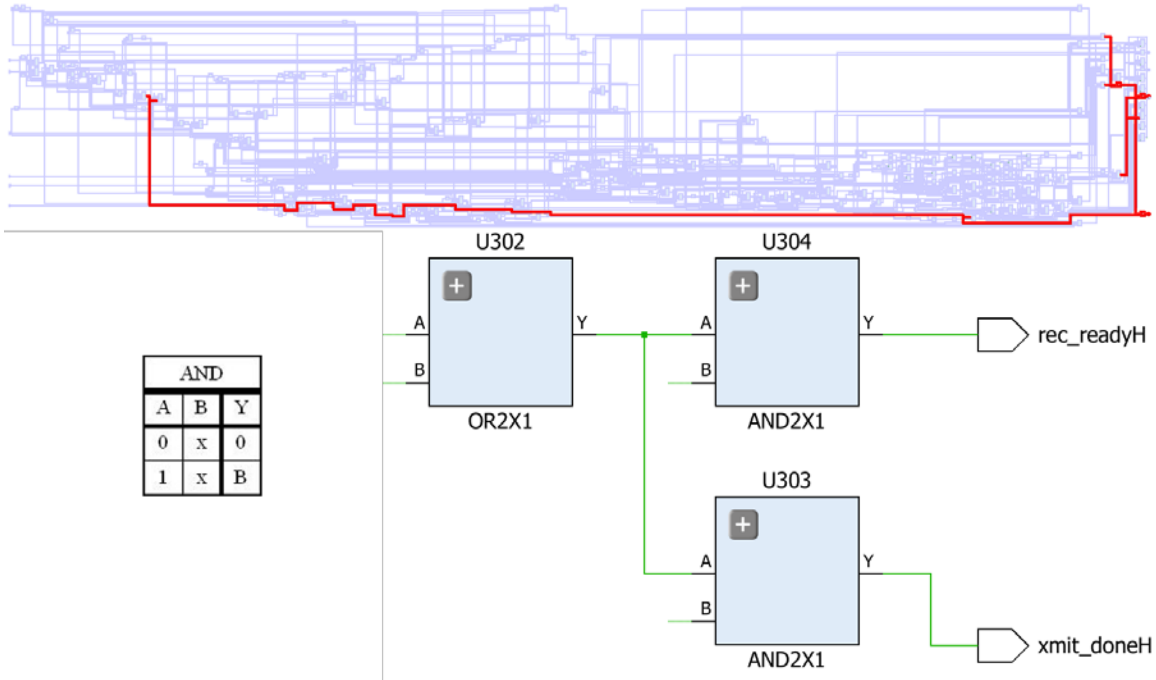
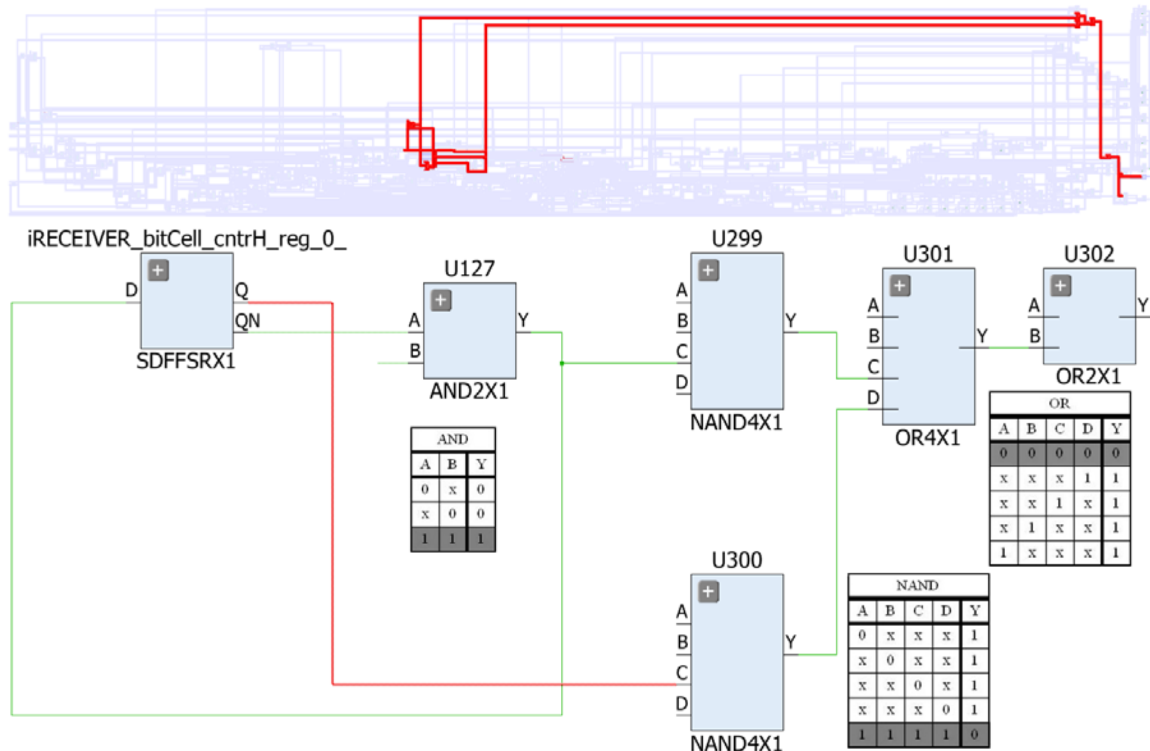


Figure 62. The functionality of `RS232-T1300`. Note that `U304` controls the `rec_readyH` output, and `U303` controls xmit_doneH. Since both are AND gates, the `iCTRL` wire can be used to force these outputs to 0_2.

6. RS232-T1400

a. Trigger

The Trojan in `RS232-T1400` cannot trigger as designed, because the triggering state requires mutually exclusive output values from a flip-flop. The relevant gates are shown in Figure 63. Wire `iCTRL` is the `Y` output of the `OR` gate `U302`. To achieve an output of `iCTRL = 0`$_2$, all of the inputs to `U302` must also be 0_2.



AND		
A	B	Y
0	x	0
x	0	0
1	1	1

OR				
A	B	C	D	Y
0	0	0	0	0
x	x	x	1	1
x	x	1	x	1
x	1	x	x	1
1	x	x	x	1

NAND				
A	B	C	D	Y
0	x	x	x	1
x	0	x	x	1
x	x	0	x	1
x	x	x	0	1
1	1	1	1	0

Figure 63. Partial schematic of the trigger mechanism of RS232-T1400. Only the gates that display the contradictory requirements are shown here. The Y output of U302 is iCTRL, which represents the activation state of the Trojan. For the Trojan to be active, this output must have a value of 0_2. Truth tables have been provided for the relevant gates. The highlighted entry in each truth table represents the output required to activate the Trojan.

Similarly, U301 must accept only 0_2 inputs, to insure that its output meets the required condition of serving as a 0_2 input to U302. Note that the C and D inputs of U301 are provided by U299 and U300, which are both NAND gates. As shown in the NAND truth table, both of these gates will need to accept only 1_2 inputs. For U300(C), this can be provided directly by iRECEIVER_bitCell_cntrH_reg_0_(Q). For U299(C), the 1_2 is provided by U127.

As an AND gate, U127 will also need 1_2 inputs. One of the inputs to U127 is iRECEIVER_bitCell_cntrH_reg_0_(QN). Note that QN is “Q not,” or the opposite of Q. Since we already established the Q output of this flip-flop as 1_2 to meet the conditions of U300, QN must be 0_2. This forces the output of U127 to 0_2, meaning that

the required Trojan activation conditions cannot be met. As a result, RS232-T1400 is incapable of altering the outputs of the circuit.

b. Functionality

The functionality of RS232-T1400 is borrowed directly from RS232-T1200. An AND gate, identified in the HDL source as U303, links the iCTRL wire to the xmit_doneH output. If the Trojan were able to be triggered, then iCTRL would be set to 0_2, as the conclusion of that triggering process. As a result, the AND gate U303 would force xmit_doneH to hold a 0_2 value as well. This will act as a denial-of-service by preventing the circuit from reporting that it is ready to transmit another message.

7. RS232-T1500

a. Trigger

The trigger mechanism of RS232-1500 is slightly different than that of RS232-T1400, but it suffers from the same flaw. The flip-flop iRECEIVER_bitCell_cntrH_reg_0_ is required to produce the same values on its Q and QN outputs. Without both of these outputs being equal to 1_2, the conditions for iCTRL = 0_2 cannot be met. However, Q and QN are, by the design of the flip-flop, guaranteed to have opposite values. Therefore, this Trojan cannot trigger.

b. Functionality

The functionality in this circuit is an exact match to that in RS232-T1000. The iCTRL wire has direct control over the xmit_doneH output and partial control over the uart_XMIT_dataH output. If iCTRL were able to hold a value of 0_2, it could prevent future messages from being provided to the RS232 circuit and potentially alter the current message by changing some 0_2s to 1_2s. A third AND gate accepts iCTRL as an input, but the output of that gate doesn't interact with any other part of the circuit.

8. RS232-T1600

a. *Trigger*

The triggering values of RS232-T1600 are shown in Table 3. The iXMIT_state_reg flip-flops are still found in the trace, but they are not directly connected to a NAND gate as they are in most other benchmarks in this group. As a result, these flip-flops to not have a definitive requirement for a 1_2 value as they do in most of the other circuits in this set.

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iRECEIVER_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_state_reg_#_		?	?	?	unknown
iRECEIVER_state_reg_#_		1	1	1	7

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 3. Table of triggering inputs for RS232-T1600. Note that inclusion in this circuit does not link directly to the XMIT_state_reg flip-flops. These source values are now fed to the Trojan by way of XOR gates, an approach that adds flexibility to their required values.

b. *Functionality*

The functionality in this circuit is borrowed from RS232-T1300. When active, the Trojan will force the outputs xmit_doneH and rec_readyH to 0_2. These outputs are necessary to coordinate message transmission and collection with the larger circuit. As long as these are held at 0_2, no new messages will be sent, and received messages will not be collected. The result is a complete denial-of-service on the RS232 circuit.

9. RS232-T1700

a. Trigger

Table 4 shows the required Q outputs for each register in the RS232-T1700 trigger. For this benchmark, RS232-T1900, and RS232-T2000, an additional requirement has been added. A new circuit input, ena, has been added, as shown in Figure 64. This input serves as the A input for all of the NAND inputs in this circuit. As a result, the trigger now includes a requirement that ena = 1_2. This will fulfill the A=1 requirement for each of the NAND gates in the inclusion. If all other requirements are met, then the Trojan will trigger.

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iRECEIVER_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_state_reg_#_		1	1	1	111_2
iRECEIVER_state_reg_#_		1	1	1	111_2

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 4. Table of flip-flop Q values required to trigger the Trojan in RS232-T1700. Note that this table does not include iRECEIVER_bitCell_cntrH. The flip-flops for that register are not found in the structure of RS232-T1700's trigger.

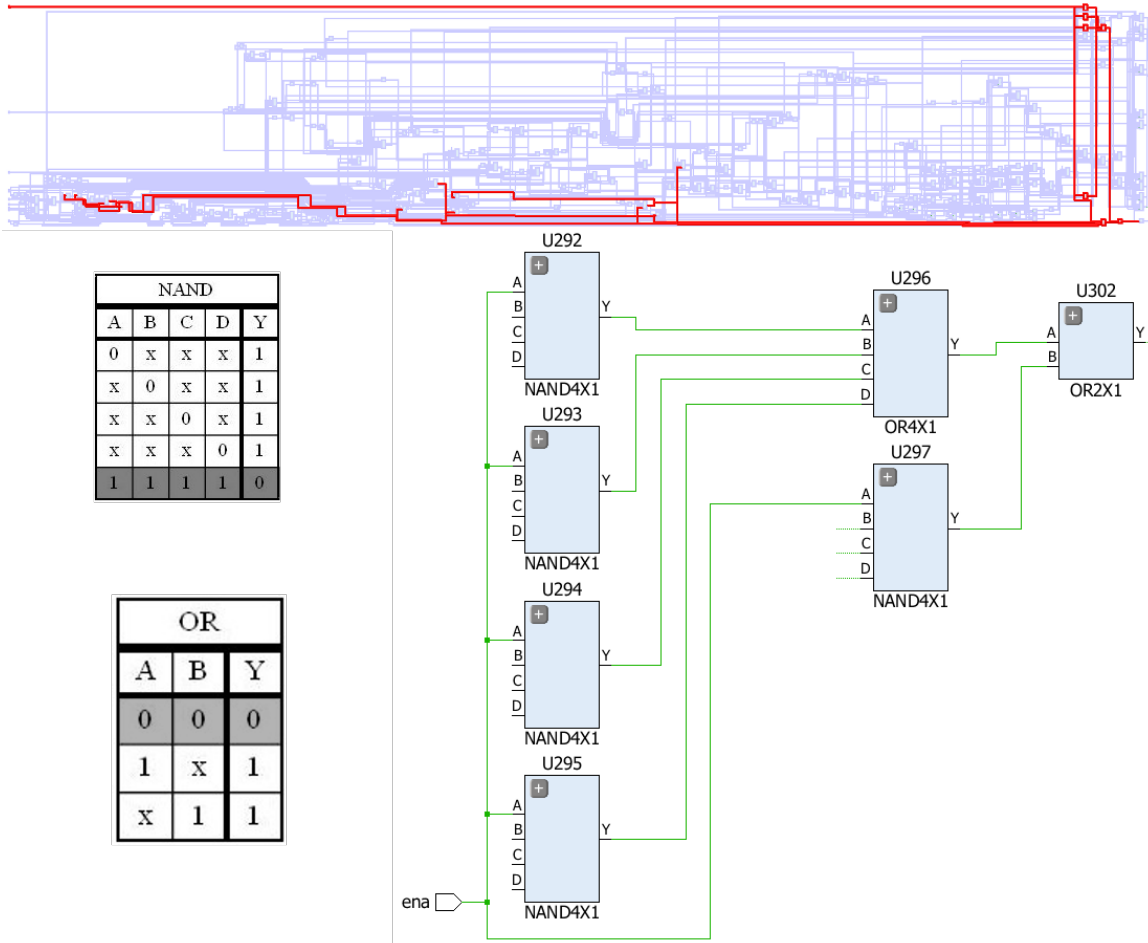


Figure 64. Partial schematic of the trigger mechanism for the `RS232-T1700` benchmark. This schematic illustrates the new `ena` input wire that was added as part of this inclusion. In this benchmark, `ena` is used as an input to each of the NAND gates in the inclusion. This replaces some of the intermediate inputs that were used to control the Trojan in other benchmarks in this set.

b. Functionality

In this circuit, `iCTRL` controls only the output signal `xmit_doneH`. Without the coordination provided by this signal, the larger architecture will not provide any new messages to be transmitted to other devices. This results in a denial-of-service attack against the transmitter portion of `RS232`.

10. RS232-T1800

The RS232-T1800 circuit is unique in this collection. The entire inclusion is composed of four gates, which have almost no interaction with the rest of the circuit.

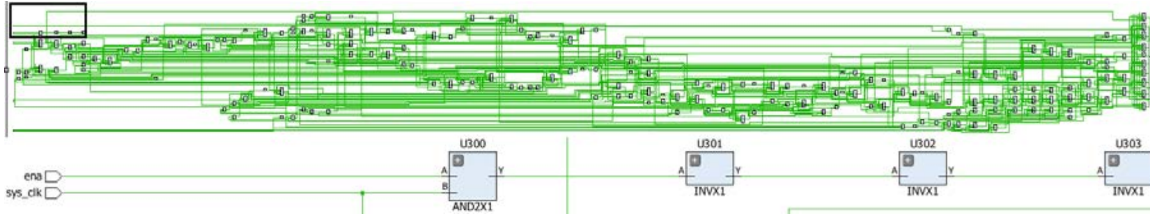


Figure 65. The complete inclusion in `RS232-T1800`. Note that the output of the final `INV` gate is not used anywhere else in the circuit. The Trojan is almost completely isolated from the rest of the circuit, sharing only the `sys_clk` input.

a. Trigger

The Trojan in this circuit is controlled by an extra input wire, `ena`, which is shown in Figure 65. In this case, `ena = 1`$_2$ represents an active Trojan. This allows the output of AND gate `U300` to change according to `sys_clk`. The value `ena = 0`$_2$ would completely disable the additional functionality of these gates.

b. Functionality

It is not clear exactly what this Trojan accomplishes. The documentation states that the Trojan will "reduce design reliability," but does not explain how. The final output of the inclusion, `sys_clk_hh`, is not used anywhere else in the circuit. Note that this Trojan does cause a group of internal wires to rapidly alternate values. This action is similar to the battery draining register used in several of the AES designs. We have been unable to determine any other potential explanation for this Trojan's functionality.

11. RS232-T1900

a. Trigger

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iXMIT_state_reg_#_		?	?	?	unknown

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 5. Table of flip-flop Q values required to trigger the Trojan in RS232-T1900. Note that the iRECEIVER conditions have all been removed from this inclusion.

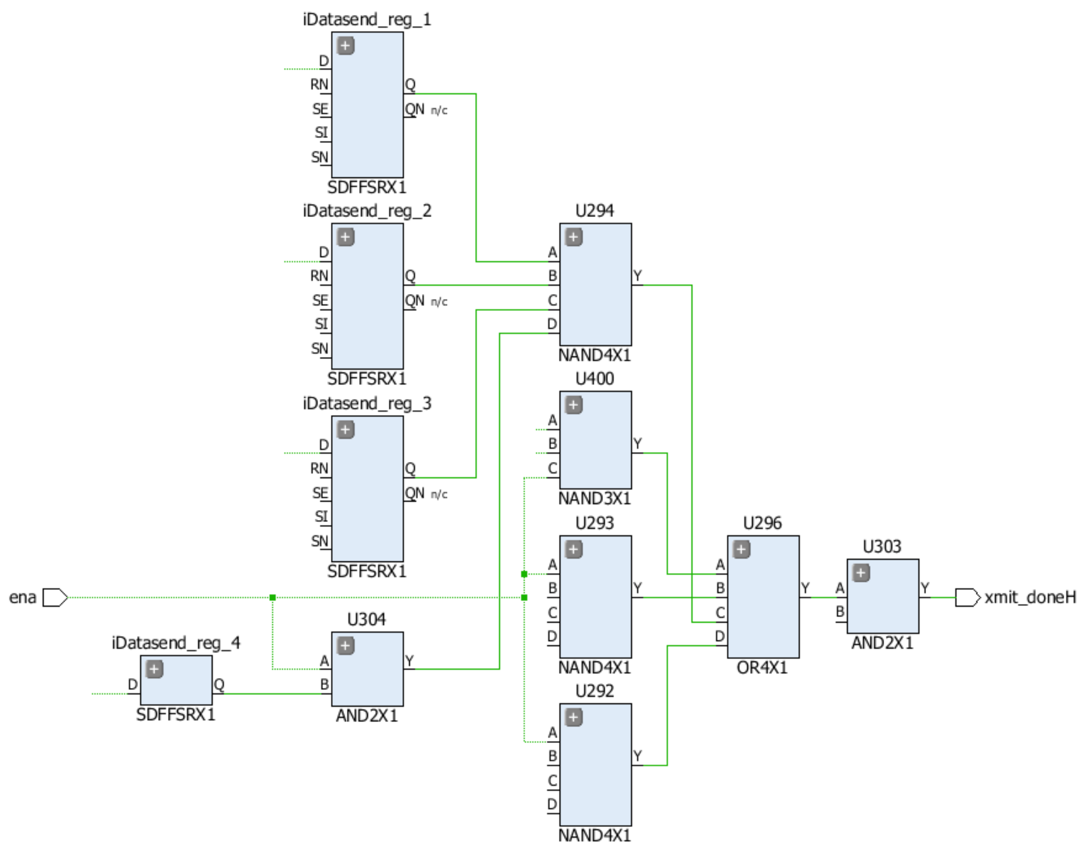


Figure 66. Schematic showing the inclusion in RS232-T1900. Note that U302 has been replaced by U296. In addition, the circuit is dependent on 1_2 values on ena and the iDatasend flip-flops.

The trigger in RS232-T1900 does not include the 2-way OR gate U302. Instead, iCTRL is provided directly by the 4-way OR gate U296, as shown in Figure 66. This gate still accepts inputs from NAND gates, but source values now include circuit input ena and the iDatasend flip-flops. Each of these extra source values needs to provide a 1_2 value to meet Trojan conditions.

b. Functionality

The Trojan in RS232-T1900 performs a denial-of-service against the transmitter portion of the circuit, just like RS232-T1200 and RS232-T1400. This is accomplished by forcing the value of xmit_doneH to 0_2. If the overall architecture follows the conventions of this circuit, then no new messages will be provided for transmission until xmit_doneH carries a 1_2. This delays the next transmission for as long as the Trojan remains active.

12. RS232-T2000

a. Trigger

Table 6 lists the flip-flop values required to trigger the Trojan in RS232-T2000. This group is supplemented by the flip-flop iDatasend_reg. To trigger the Trojan, iDatasend_reg must have a Q output of 0_2 at the same time that all of the listed flip-flop conditions are met, and the input ena carries a 1_2 signal.

Register	3	2	1	0	Value
iXMIT_bitCell_cntrH_reg_#_	1	1	1	0	0xE
iXMIT_bitCount_reg_#_	1	1	≠[0]	≠[1]	13 or 14
iXMIT_state_reg_#_	■	1	1	1	111_2

Register	7	6	5	Value
iXMIT_xmit_ShiftRegH_reg_#_	1	1	1	0xE0 to 0xFF

Table 6. Triggering inputs for RS232-T2000. Note that the receiver-side logic has been completely removed from this inclusion's trigger. Instead, the B input of U302 is provided by the flip-flop iDatasend_reg

There are two possible input combinations that will assign a value of 0_2 to iDatasend_reg's Q output. These combinations are shown in Table 7. Note that once a value is assigned to Q, it will remain assigned until an input configuration is seen that forces the value to change.

	First	Second
sys_clk	rising	rising
ena	1	1
sys_rst	1	1
iRECEIVER_rec_datSyncH_reg(Q)	0	x
test_se	0	1
iRECEIVER_par_dataH_reg_7_(Q)	x	0

Table 7. Table listing the source values required to set iDatasend(Q) = 0_2. Wires sys_clk, ena, sys_rst, and test_se are circuit inputs. The other values identified here are flip-flop outputs.

b. Functionality

When activated, RS232-T2000 conducts a denial-of-service against the transmitter and receiver portions of the circuit. This is accomplished by forcing both xmit_doneH and rec_readyH to hold a value of 0_2 for as long as iCTRL has a value of 0_2. Note that this is the same mechanism used in RS232-T1300 and RS232-T1600.

V. CONCLUSION

A. SUMMARY

Of the benchmarks we have examined, the AES benchmarks are the best suited for use by researchers with limited prior exposure to the resources at the Trust-Hub website [8]. This is based on several distinct features of this benchmark set. First, each benchmark's documentation explicitly identifies the triggering condition for that benchmark. The documentation also describes what effect the Trojan will have when it has been activated.

Second, each AES Trojan is designed in a modular structure, with the trigger and functionality each being written in a separate file. This feature greatly simplifies static HDL analysis. Note that there are a few exceptions, most notably `AES-T1800` and `AES-T1900`, each of which merges the trigger and functionality into a single module. Also note that the Trojans in these benchmarks actually operate in isolation from the `AES_128` module. These Trojans accept the same input busses as the Trojan-free circuit, but they do not actually alter existing output busses. Instead, they provide alternate channels for the leakage of secret information.

The next significant feature of this benchmark set is the inclusion of a Trojan-free implementation of `AES_128` with each benchmark. This circuit can serve as a 'control' circuit for simulation purposes, allowing a researcher to compare the activity of a Trojan-inclusive circuit with that of a Trojan-free circuit.

Finally, each of the AES benchmarks includes a selection of test benches. These test benches have been written so that they actually provide the input values required to activate the combinatorial and sequential triggers among the AES Trojans. Note that none of the provided test benches will cause the activation of the counter-based AES Trojans to activate.

The `basicRSA` benchmarks are somewhat less easy-to-use than the AES benchmarks. However, the `basicRSA` benchmarks do share some of the advantages of the AES benchmarks. The benchmark documentation for each of these circuits does

describe the activation condition and functionality of the Trojan. Each benchmark also includes a dedicated test bench designed to trigger the Trojan in that benchmark. Note that while the benchmark archives do not actually include a Trojan-free implementation of the circuit, one can be found at the opencores.org website.

The key distinction between the AES circuits and the `basicRSA` circuits is found when we attempt to perform static analysis on the HDL code. The `basicRSA` benchmarks do not assign specific modules or files to the Trojan aspects. This makes them slightly more difficult to analyze, but also allows them to be more easily merged with the existing circuit. The `basicRSA` circuits actually alter the function of the core circuit, causing the final output bus to produce different values, including the direct leakage of the private exponent. Note that the difficulty of analyzing the HDL is mitigated by the liberal use of comments in `rsacypher.vhd`. The HDL defining the inclusion is specifically identified in comments within each instance of this file.

It will be more difficult for a researcher to develop familiarity with the RS232 benchmarks. The difficulties in these benchmarks are the result of synthesis challenges and limited documentation. We have made an effort to address these difficulties here in this thesis. Note that we have provided a method for constructing a Trojan-free implementation of the RTL version of RS232, but we have been unable to find a Trojan-free circuit to use in conjunction with the gate-level benchmarks. Note that none of the RS232 benchmarks have been provided with a test bench. We have written test benches that correctly trigger Trojans in the RTL set of benchmarks. These test benches can also be used to simulate normal operation of any of the gate-level implementations of RS232. Note that we have been unsuccessful in our efforts to develop a test bench that will trigger the Trojans in any of the gate-level implementations.

Five of the 10 RTL-based benchmarks accurately identify their triggering condition. `RS232-T600` and `RS232-T900` list an incorrect combination, and `RS232-T100`, `RS232-T200`, and `RS232-T800` provide an incomplete description of their triggering conditions. In particular, the documentation for each of these three benchmarks

defines the number of signals that control the Trojan trigger, but does not specifically identify what signals are used, or what values they need to hold.

The documentation in the gate-level RS232 benchmarks doesn't discuss source values for the Trojan trigger mechanism at all. Instead, the documentation identifies the activation probability for each Trojan, and shows statistics from tests performed by the benchmark authors. The documentation also lists added lines of HDL, which proved useful in tracing through the circuit diagrams in order to determine the values required to activate each Trojan.

The RS232 benchmarks also require additional editing in order to synthesize correctly in Vivado. The files in the RTL version of these benchmarks contain an `include` directive with a hard-coded path that needs to be rewritten in order for the benchmark to be used on a computer other than the original author's. The gate-level benchmarks are based on a library that is not available for use in Vivado. In conjunction with this thesis, we have provided a substitute library that provides a logical implementation of each module required by the RS232 gate-level benchmarks.

B. FUTURE WORK AND LESSONS LEARNED

In this thesis, we have analyzed 46 of the 92 benchmarks available on `Trust-Hub` [8]. We focused this thesis on those that we were able to provide an in-depth analysis of, with a preference for the largest benchmark sets, namely AES and RS232. It would be helpful for future researchers to complete this study, providing a full, in-depth description of each of the remaining benchmarks. Note that many of these benchmarks are gate-level designs that will be easier to synthesize using the libraries `lib_90.v` and `lib_180.v`.

Future researchers should also conduct an evaluation of existing Trojan detection and mitigation techniques against the benchmarks in this collection. The purpose of this study would be to evaluate the usefulness of these Trojans as common benchmarks. If existing techniques can be effectively applied against these Trojans, then new techniques can be evaluated against them as well. The common frame of reference will provide a more meaningful tool for evaluating detection and mitigation techniques against one

another. If there are difficulties in applying a particular detection method, then the results of that investigation can be used to improve the structure of the available benchmarks.

In addition to conducting analysis against the existing benchmarks, researchers should consider expanding the collection. Only 15 different circuits are represented among the benchmarks at the Trust-Hub website [8], counting the RTL and gate-level versions of RS232 separately. The `vga_lcd` circuit is represented by only a single benchmark. We propose that the collection be expanded to include a wider array of benchmarks. Based on our research here, we would like to propose some structural elements that can be incorporated into future benchmarks.

C. ELEMENTS OF FUTURE BENCHMARKS

This section is meant to serve as a high-level sketch of future benchmarks that can be added to the collection at [8], based on our experiences in simulating the circuits discussed in this thesis. It is our hope that the incorporation of features discussed here will make it easier for new researchers to use this collection as a common source of standard benchmarks when conducting research into Trojan detection and mitigation.

The first benchmark feature we will discuss here is documentation. Based on our experiences with RS232 benchmarks, we propose that documentation of all future benchmarks include, at minimum, the following features:

- A short description of the Trojan-free circuit, including its purpose, inputs and outputs.
- A detailed list of all register and input values required to activate combinatorial or sequential triggers, if applicable. If the trigger is based on a counter, the description of the trigger should identify whether the counter counts clock cycles or circuit operations.
- A description of the Trojan's impact on the circuit after activation. Ideally, this description will include a listing of expected values based on a test bench that is provided as part of the benchmark archive.

- If a Trojan-free implementation of the circuit is not already provided in a benchmark archive, the documentation should identify the source of the original circuit. This may be a website like `opencores.org`, or an explanation of the use of tools for automatic HDL generation.

The structure of future Trojans will vary dramatically, but we propose that the trigger and the functionality for each Trojan be written as separate modules. These modules will allow a new researcher to more quickly perform a static analysis of the changes to the HDL between different benchmarks. Note that not all HDL changes will be part of these modules. The AES Trojans discussed in this thesis treated the Trojans as completely separate circuits, and as a result, the `aes_128` module remained unchanged. While this technique produces the simplest analysis, it limits the creation of Trojans that interfere with circuit outputs or make use of internal registers. Therefore, we suggest that the trigger and functionality modules be instantiated within the modified circuit. This represents a compromise between the versatility of the Trojan and the ease of analysis for a researcher new to the benchmarks.

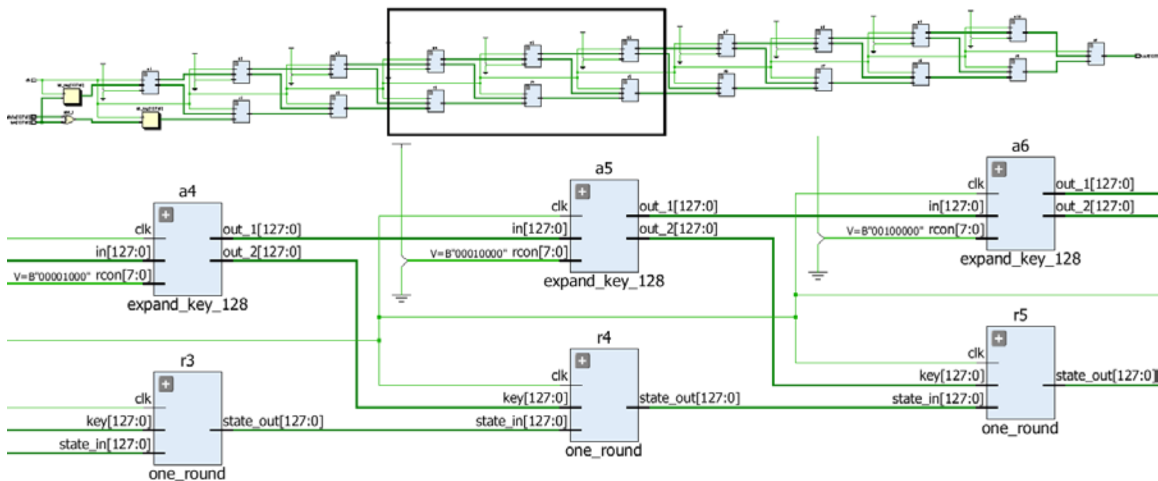


Figure 67. A portion of the Trojan-free AES circuit diagram. We will use this to identify specific internal wires and busses that can be used as inputs to a `Trojan_trigger` module or outputs form a `TSC` module, similar to those used in earlier AES benchmarks.

As an example, we propose a high-level sketch of an AES Trojan that is incorporated into the AES circuit. Examine Figure 67. This figure displays 3 rounds of the AES encryption process. We propose the addition of two modules: `Trojan_trigger` and `TSC`.

The `Trojan_Trigger` module will accept the bus `state_out` of module `r4` as an input and produce a single-bit wire `tj_trig` as an output. Internally, `Trojan_Trigger` will compare bus `state_out` of module `r4` to the preselected value `0xAAAA…AAAA`. If these values are identical, then `tj_trig` will be set to 1_2. Because the AES circuit does not include a reset signal, Trojan trigger will also include a 32-bit counter that increments every clock cycle while `tj_trig` = 1_2. When this counter reaches a value of `0xFFFFFFFF`, `tj_trig` will be set to 0_2, and the counter will be reset to `0x0`.

Module `TSC` will accept `tj_trig` and bus `state_out` of `r4` as inputs, and the output of `TSC` will act as bus `state_in` of module `r5`. While `tj_trig` = 0_2, `TSC` will feed bus `state_out` of module `r4` directly to bus `state_in` of module `r5`, exactly as shown in Figure 67. When `tj_trig` = 1_2, `TSC` will instead use a chosen plaintext value as bus `state_in` of module `r5`.

The proposed Trojan serves primarily as an attack against reliability. After a plaintext/key combination produces the correct output from module `r4`, future outputs from the circuit will be based on the encryption of the attacker's chosen plaintext. Note that the attacker might be able to use this plaintext and the resulting ciphertext to determine the value of roughly half of the intermediate round keys used during the encryption. If the key expansion algorithm is known, the attacker may even be able to use this information to determine the original value of circuit input `key`.

The Trojan discussed above interferes with the internal operation of the module `aes_128`, but also allows the benchmark authors to implement the trigger and functionality as dedicated, easily-analyzed modules. We believe that this will be a valuable aid to researchers who do not have a pre-existing familiarity with the benchmarks.

APPENDIX. RESOURCES

As part of this thesis, we have created several Verilog files that should assist the reader in simulating the benchmarks in the RS232 set. These additional resources can be found at https://calhoun.nps.edu/handle/10945/45406. At this URL, you will find an archive named `Slayback_thesis_resources.zip`. Within that archive, we have supplied 5 Verilog files and a readme explaining the purpose of each file.

The files `lib_90.v` and `lib_180.v` are library files that provide definitions for several modules required by the benchmarks `RS232-T1000` through `RS232-T2000`. None of these benchmarks will synthesize unless you include the appropriate module library. To do this, follow the project creation process as described in Chapter III. When selecting source files to build the project, add `uart.v` from the benchmark archive, then add one of these libraries. Use `lib_90.v` with the 90nm version of each benchmark and `lib_180.v` with the 180nm version. Once the project has been built with these files, synthesis and elaboration can be conducted without error.

The remaining files are test benches. Section III-D explains how to import a test bench into an existing project. These test benches were designed as baseline test benches for the RTL versions of RS232. For receiver-based Trojans, such as those in `RS232-T100` and `RS232-T200`, use the test bench `test_rs232_rec.v`. This file provides 10 input sequences to the receiver, but none to the transmitter. For transmitter-based Trojans, such as `RS232-T300` and `RS232-T500`, use `test_rs232_rec.v`. For consistency, this test bench provides the 10 values from `test_rs232_rec.v` as 1-byte inputs. Some changes to this sequence may be required to trigger an individual Trojan.

The last test bench, `test_rs232_400.v`, provides 4 inputs each to the receiver and transmitter modules. This file can be used to conduct experiments with Trojans that depend on simultaneous activity in both modules.

Note that we have used these test benches on the gate-level RS232 benchmarks. We have been able to simulate normal RS232 activity, but we have been unable to trigger

the Trojan contained in any of the benchmarks from `RS232-T1000` through `RS232-T2000`.

LIST OF REFERENCES

[1] J. Markoff, "Old trick threatens the newest weapons," *New York Times*, p. D1, Oct. 27, 2009.

[2] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[3] X. Wang, M. Tehranipoor and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, Jun. 2008, pp. 15–19.

[4] R. Karri, J. Rajendran, K. Rosenfeld and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware Trojans," IEEE Computer, vol. 43, no. 10, pp. 39–46, Oct. 2010.

[5] F. Wolff, C. Papachristou, S. Bhunia and R. S. Chakraborty, "Towards Trojan-free trusted ICs: Problem analysis and detection scheme," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, Mar. 2008, pp. 1362–1365.

[6] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, San Diego, CA, Jun. 2011, pp. 67–70.

[7] M. Banga and M. S. Hsiao, "A region based approach for the identification of hardware Trojans," in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, Jun. 2008, pp. 40–47.

[8] Trust-HUB Resources: Benchmarks. [Online]. Available: https://www.trust-hub.org/resources/benchmarks. Accessed Jun. 19, 2015

[9] S. Kutzner, A. Y. Poschmann and M. Stöttinger, "Hardware Trojan design and detection: A practical evaluation," in *Proceedings of the Workshop on Embedded Systems Security (WESS)*, Montreal, Canada, Sept. 2013, pp. 1–9.

[10] Y. Alkabani and F. Koushanfar, "Consistency-based characterization for IC Trojan detection," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, Nov. 2009, pp. 123–127.

[11] D. McIntyre, F. Wolff, C. Papachristou, S. Bhunia and D. Weyer, “Dynamic evaluation of hardware trust,” in *Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, San Francisco, CA, Jul. 2009, pp. 108–111.

[12] J. Li and J. Lach, “At-speed delay characterization for IC authentication and Trojan horse detection,” in IEEE International Workshop on Hardware-Oriented Security and Trust (HOST), Anaheim, CA, Jun. 2008, pp. 8–14.

[13] M. Hicks, M. Finnicum, S. T. King, M. Martin and J. M. Smith, “Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically,” in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, Oakland, CA, May 2010, pp. 159–172.

[14] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang and Y. Zhou, “Designing and Implementing Malicious Hardware.” in *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, San Francisco, CA, April 2008, pp. 1–8

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California