CSE 145/237D Embedded Systems Project

IFT for Trojan detection

By

Santosh Krishnan(A53071285) Darren Eck(A12490850)

Assisted By

Vinnie Hu

Abstract
The reliance on computer systems increases every year. As the amount of information collected, stored, and controlled by computer systems increases, the demand to secure these systems and the critical data they carry increases as well. New vulnerabilities are emerging through malicious modifications to the design or production of circuits. These modifications, classified as “hardware trojans,” can be very difficult to detect. By applying Gate-Level Information Flow Tracking (GLIFT) techniques to hardware design files, our aim is to understand the flow of sensitive data through a system and make arguments about the presence of potential hardware trojans within a system. We achieve a finer granularity and greater accuracy in detecting flows of information than is possible with traditional information flow tracking, while still utilizing traditional information flow tracking to reduce processing time where information flow is not possible. We apply the solution and show the possible trojans with combinational logic designs.

Introduction

The hardware trojan is a relatively new vector in which attacks may be launched to compromise the security or proper function of a computer or electronic system. These trojans are changes to an integrated circuit or system that can have potentially catastrophic consequences. A trojan can provide an opening from which a third party can gather secure data, allow unauthorized control or code execution, send false output signals, or simply disrupt or destroy the operation of the chip at critical moments. The trusted operation of these chips is becoming more critical as systems, such as in military and security equipment, incorporates more and more circuits, often produced overseas. For example, in 2007, an Israeli airstrike went undetected in Syria. The failing of the Syrian air security systems is rumored to have been caused by a hardware trojan in the foreign bought equipment [4].

As the number of transistors increases on a single chip, it also becomes increasingly easier to hide these trojans within the mass of circuitry. A small modification becomes incredibly subtle. One group designed and implemented a processor that supported general security attacks through a backdoor, while only adding a measly 1,300 gates to a processor already containing hundreds of thousand of gates and nearly 2 million transistors [3].

The production and distribution of these IC chips goes through a number of steps and facilities before completion. Much of the process ends up in various facilities around the world, as the demand surpasses what can be manufactured locally. Counterfeit or malicious chips have been found and seen before from various countries. Currently, the main method to ensure the integrity of chips in secure systems is by using a network of trusted foundries and suppliers [4].

Hardware trojans may also be injected into the design phase of a circuit. This can occur through the use of intellectual property (IP) cores in a design. An IP core is a reusable unit of logic or part of a circuit that can be added and incorporated into other systems. Companies will produce IP cores to be distributed to others and incorporated into designs in order to save them time and money. However, these IP cores can contain hardware trojans that go undetected and are distributed on a potentially global scale.

Our aim is to employ the techniques of gate-level information flow tracking (GLIFT) to help detect harmful flows of information through a secure system in order to detect potential trojans within a system. GLIFT is a method of augmenting a design with additional tracking, or shadow, logic that is responsible for tracking the trust of individual bits throughout a system.

Many modern systems rely on strict information flow policies to manage the security and determine where throughout a system critical information can leak. Often times, these solutions are implemented in software, such as in the programming languages or even at the OS level. IFT is often employed because of its efficiency when used to prevent harmful flows of information in software based attacks [1]. IFT has also been implemented at the instruction set level, and more recently, at the gate-level [1]. By going to the gate-level for IFT, we can achieve a finer precision of accurate information tracking to reduce false positives of potential security vulnerabilities. We use GLIFT to detect trojans at the RTL level.

We implemented and simulate both a conservative IFT model, as well as a precise model that can only be achieved at the gate level. By first using the conservative model, we can gain an understanding of where vulnerabilities and potential trojans may exist with an RTL design. We use the results of the conservative simulation to select certain paths within the system to replace with a precise model. This allows much less additional logic and processing time than an entire design augmented with precise gate-level tracking, while avoiding false negatives on harmful flow detection. Then, we simulate the precise model with random inputs to gain an idea of the general flow of information. We can then use that information to make arguments about security or the presence of hardware trojans.

Technical Material

- <u>IFT and shadow logic</u>

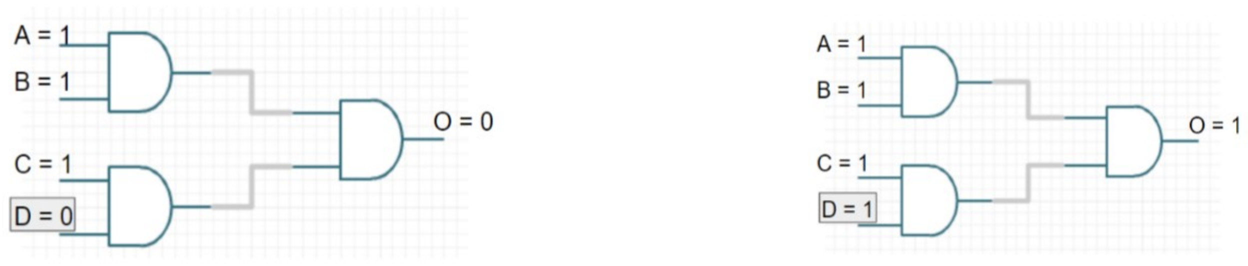


Figure 1 : information flow.

Given a netlist with i inputs and j outputs, one of which is a secret key ‘k’ we say that information flows from the key ‘k’ to the output if a change in the secret key input ‘k’ results in a change in the output whilst keeping the other inputs exactly the same In figure 1, a change in input D from 0 to 1 results in a change in the output O so we say that information from D leaks to the output O.Traditionally the solution to this problem is obtained by solving the boolean satisfiability problem (also called SAT problem) which tries to determine possible input conditions that result in a given output. This problem is NP complete meaning that it doesn't scale well there is no standard way to solve this problem accurately for large designs.

We , however use this concept and take it to the gate level. For example in figure 1, If, by some modelling, we can show that information flows from D to OP1 and from OP1 to O, then we can conclude that information flows from D to O. Solving this problem now takes much less time. The complexity of this algorithm is equal to any graph traversing algorithm.

The granularity in the algorithm is achieved by modelling each gate in the netlist to show information flow at the gate level. This modelling is called shadow logic modelling. For our analysis, we used two types of modelling for our shadow logic. They are the conservative model and the precise model.

- Conservative and the precise model

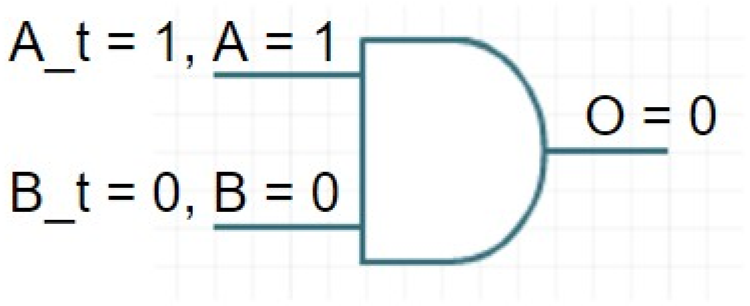
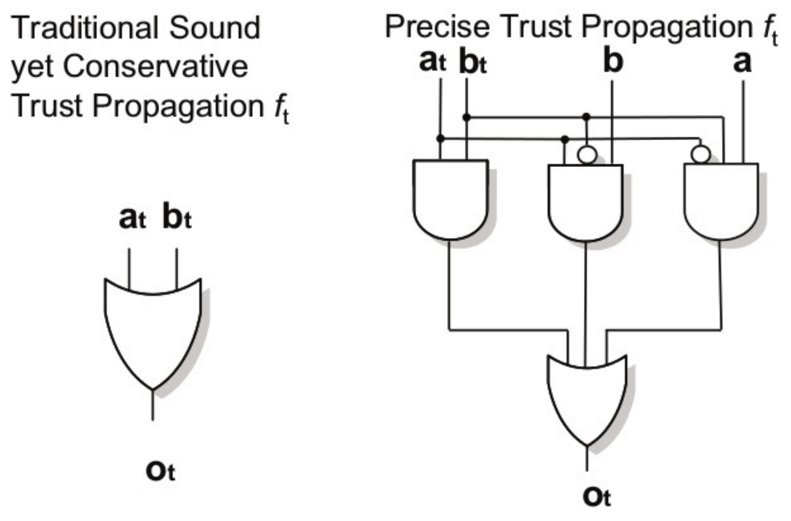


Figure 2: And gate with Input A being tracked

Figure 2 represents the basic concept of the conservative model of the shadow logic.The and gate has two inputs A and B and a output O. We want to see whether information from input A alone flows to the output. We indicate this fact by making A_t = 1 and B_t = 0 and say that input A is being tracked. If we want to see whether information from either input A or input B flows to the output we make A_t = 1 and B_t = 1 and say that inputs A and B are both tracked. The conservative model states that if an input is being tracked, information flows from the input being tracked to the output irrespective of the actual inputs. Therefore, as shown in figure 2, according to the conservative model, information flows from input A to O.

The precise model, however states that information flows from an input to the output if a change in that input results in a change in that output. So , in figure 2, since input A is being tracked, a change in input A from 0 to 1 does not change the output. So we say that according to the precise model, Information does not flow to the output.


Traditional Sound yet Conservative Trust Propagation f_t
a_t b_t
o_t
Precise Trust Propagation f_t
a_t b_t b a
o_t

A	B	A_t	B_t	O (consv)	O(Precise)
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	1	1
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1

a b

Figure 3 a. logical representation of the conservative and precise model b. truth table for the conservative and precise model.

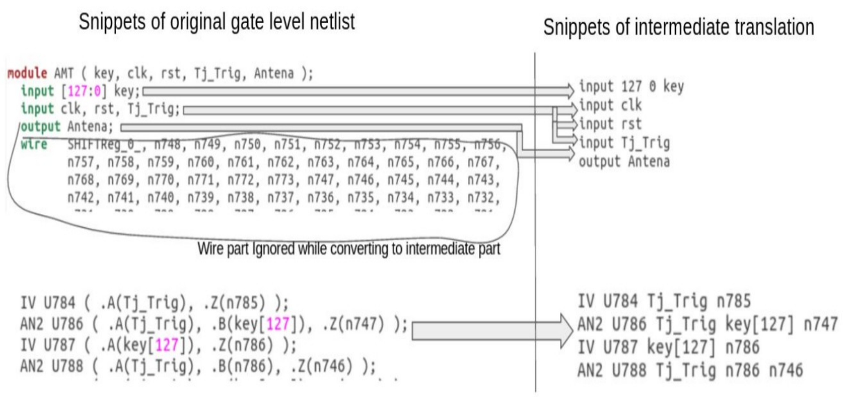
Figure 3a gives the logical representation of the conservative and precise model and figure 3b gives the truth table for the conservative and precise model. On comparing the two truth tables we see that there are some cases which show that information flow happens according to the conservative model and not so according to the precise model (when A=0, B=1,A_t= 0, B_t=1) but never vice versa. This is true for all basic gates. Thus we can say that in a netlist,if according to the precise model, there exist a path through which information flows from an input to the output, then the conservative model will also show information flow through that path. But the same isn't true vice-versa.

Gate	Conservative	Precise
$Y = A \cdot B$	$Y = A \cdot B$ $Y_t = A_t + B_t$	$Y = A \cdot B$ $Y_t = B \cdot A_t + A \cdot B_t + A_t \cdot B_t$
$Y = A + B$	$Y = A + B$ $Y_t = A_t + B_t$	$Y = A + B$ $Y_t = \overline{B} \cdot A_t + \overline{A} \cdot B_t + A_t \cdot B_t$
$Y = \overline{A}$	$Y = \overline{A}$ $Y_t = A_t$	$Y = \overline{A}$ $Y_t = A_t$

Figure 4: Conservative and precise model for some simple gates.

- <u>Our algorithm</u>

  Before we dive into the algorithm, We need to understand what are the inputs to the algorithm. We have two inputs: an input file which has the values of all the inputs to the netlist and the input that we want to track i.e. the input that we want to observe whether it leaks out its secret information to the output and the other is a simplified version of the netlist file called intermediate netlist format.


Snippets of original gate level netlist
Snippets of intermediate translation
Wire part Ignored while converting to intermediate part

```
module AMT ( key, clk, rst, Tj_Trig, Antena );
  input [127:0] key;
  input clk, rst, Tj_Trig;
  output Antena;
  wire   SHIFTReg_0_, n748, n749, n750, n751, n752, n753, n754, n755, n756,
         n757, n758, n759, n760, n761, n762, n763, n764, n765, n766, n767,
         n768, n769, n770, n771, n772, n773, n747, n746, n745, n744, n743,
         n742, n741, n740, n739, n738, n737, n736, n735, n734, n733, n732,

  IV U784 ( .A(Tj_Trig), .Z(n785) );
  AN2 U786 ( .A(Tj_Trig), .B(key[127]), .Z(n747) );
  IV U787 ( .A(key[127]), .Z(n786) );
  AN2 U788 ( .A(Tj_Trig), .B(n786), .Z(n746) );
```

```
input 127 0 key
input clk
input rst
input Tj_Trig
output Antena

IV U784 Tj_Trig n785
AN2 U786 Tj_Trig key[127] n747
IV U787 key[127] n786
AN2 U788 Tj_Trig n786 n746
```

Figure 5: the conversion from the netlist to the intermediate netlist format.

We wrote a tcl script to achieve this conversion to the intermediate netlist format. From figure 5, we see that we handled arrays-single bits and input-output pairs separately. We also removed all extra information in the netlist description to simplify processing.

Our algorithm begins by taking to two inputs files mentioned above and building our graph. Next we set the model to be conservative and then populate our graph and our shadow logic. Notice that we use the conservative model for our shadow logic.

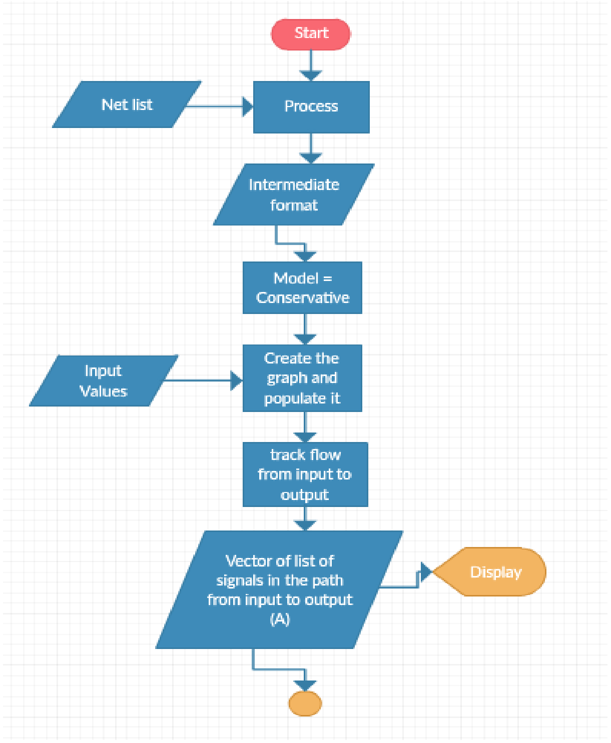

Start
Net list
Process
Intermediate format
Model = Conservative
Input Values
Create the graph and populate it
track flow from input to output
Vector of list of signals in the path from input to output (A)
Display

Figure 6: Part 1 of our algorithm.

The population process happens as follows.

- We first get all the instances that are connected to the inputs. These instances are pushed onto a queue called live instances. We also mark all the inputs as available.
- We then pop the first element in the queue. That instance is our current instance.
- If all the inputs of the current instance is available, we find the outputs of the current instance and its value. We also set a check bit to 1. We also set the output of the current instance as available.
- If the check bit is 1, We proceed to find the outputs of the current instance and the instances to which the output of the current instance go to. We push those instances into the live instance queue
- We repeat this process till the queue is empty.

Once we have populated the graph, we then proceed to find all the paths through which information from the tracked input leaks out to the output. This vector of list of signals in the path of information flow is called vector A. To find the list of paths, we use a modified depth first search algorithm where the modification is that we go deeper only if the information flows from the inputs of that gate to the output. We also have a backtracking mechanism if we find that information doesn't propagate to the output of the gate under question.

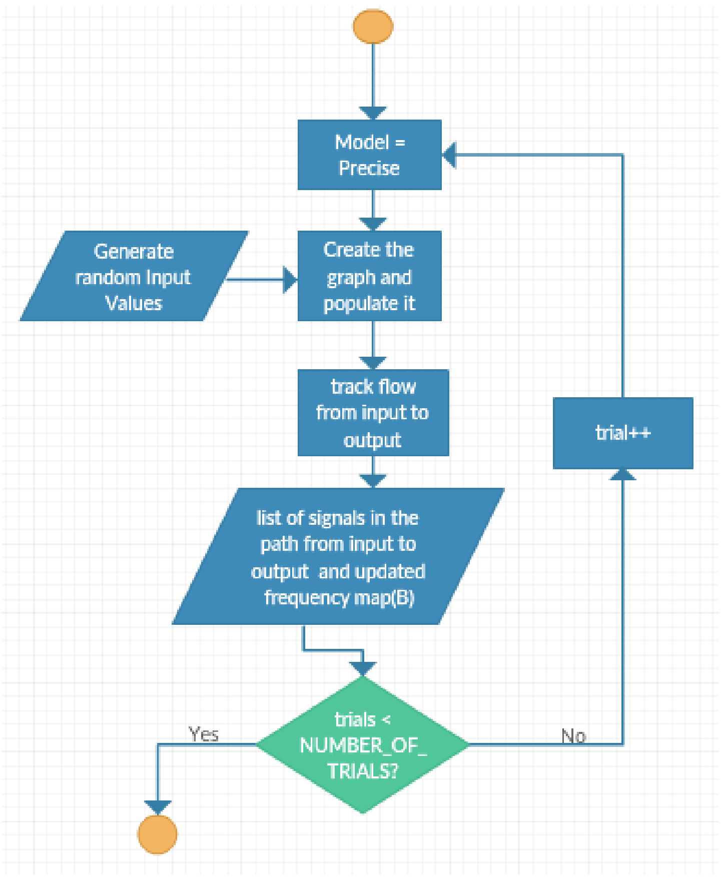

Model = Precise
Generate random Input Values
Create the graph and populate it
track flow from input to output
trial++
list of signals in the path from input to output and updated frequency map(B)
Yes
trials < NUMBER_OF_ TRIALS?
No

Figure 7: Part 2 of our algorithm:

Figure 7 represents the flowchart of the second part of our algorithm. We start by setting the model to be precise. As mentioned before as the output of the shadow logic using the precise model depends on both the actual inputs and the inputs being tracked, different values to the inputs of the netlist will yield different paths through which information flows from the tracked input to the output if there exist such paths in the first place. It therefore makes more sense to see how many times a path gets activated and information flows through it to the output. We do this by creating a map of paths that show how many times they get activated. This map is called the frequency map B. Once we set the model to be precise we, generate a set of random input values and use these inputs to populate our graph. We then find the paths through which information from the tracked input leaks out to the output and update the frequency map. The method to populate our graph and to find the paths through which information flows to the output are the same as that explained in the first part of the algorithm. We repeat the two steps for a sufficient number of times. The number of times those steps are repeated is n.

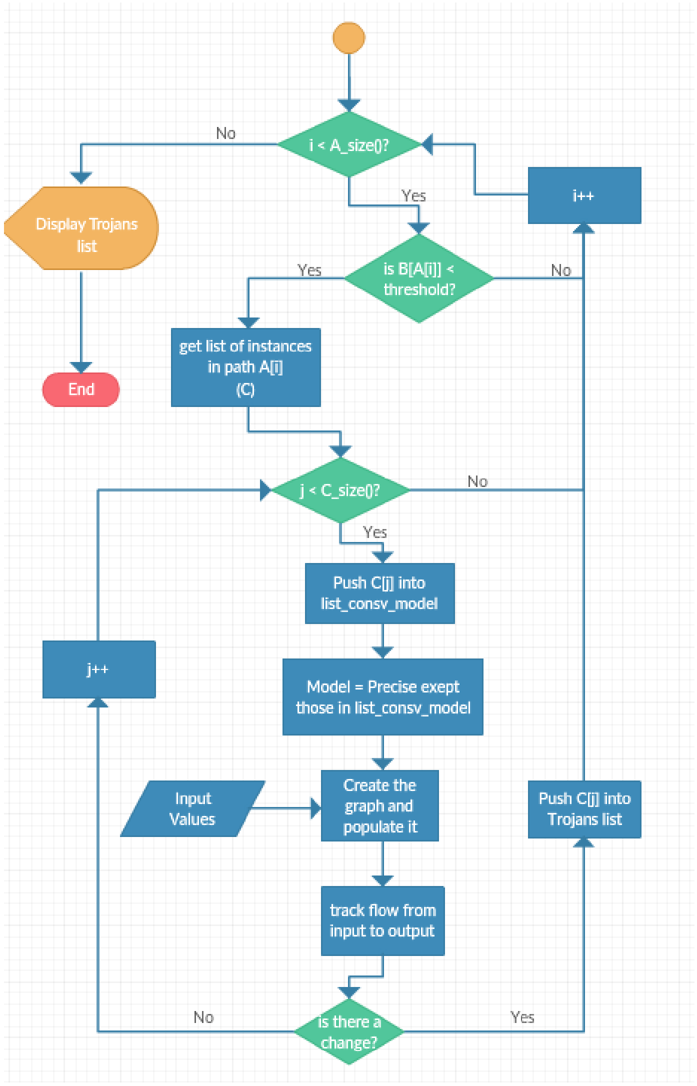

i < A_size()?
No
Yes
Display Trojans list
i++
is B[A[i]] < threshold?
Yes
No
get list of instances in path A[i] (C)
End
j < C_size()?
No
Yes
Push C[j] into list_consv_model
j++
Model = Precise exept those in list_consv_model
Input Values
Create the graph and populate it
Push C[j] into Trojans list
track flow from input to output
No
is there a change?
Yes

Figure 8: Part 3 of our algorithm.

At this point we have the vector (A) of paths of information flow from the tracked input to the output using the conservative model and the frequency map (B) consisting of paths of information flow from the tracked input to the output and the number of times that path gets activated. We decide a threshold 'd' which equals. $d=\frac{n}{2i}+1$ Wherei is the number of inputs to our netlist.Now looking at the path in A, if the value field of that path in the map B is less than or equal the threshold, we say that a trojan could exist in that path. We find the instances on that path. One by one we set each instance on that path to be conservative and keeping shadow logic model for the instances in the netlist as precise, we find whether information flows through that path to the output. Ther moment we find that information flows to the output, we conclude that the instance under consideration is a trojan. This process is repeated for all paths in A.

We demonstrated this algorithm on some benchmarks given to us by Vinnie. We also tested our algorithm on some of our own benchmarks given below. We have also marked the trojans that our algorithm managed to detect.

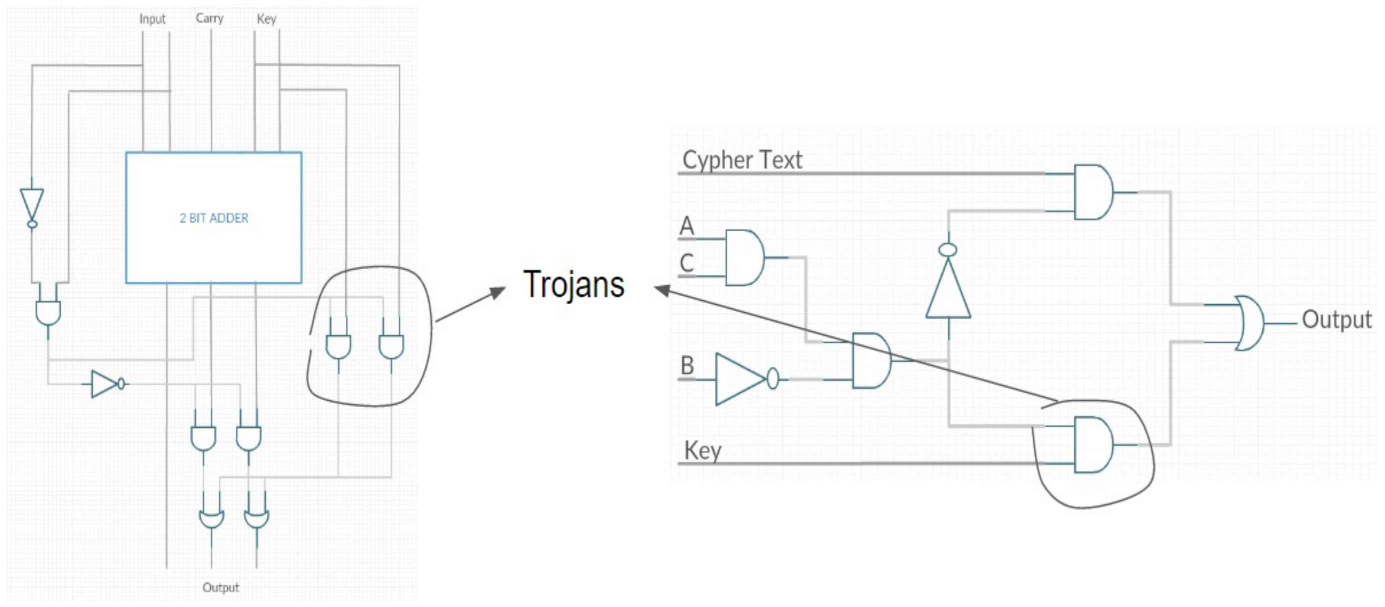


Figure9: some of our benchmarks with the trojans

Milestones

The following is a list of the milestones at the mid-quarter update.

- Parse and preprocess RTL netlist

The first step was to parse and process the RTL netlist into a format more easily processed by the main tool. This simply meant stripping unnecessary information from the list and arranging it in a way that would be able to be easily parsed by the main tool.

- Create graph

From the preprocessed netlist, we created a graph to represent and simulate the components of the design into a graph to be processed by the computer. The nodes of the graph represented the various components in the RTL netlist, and the edges represented the wires that connected the components.

- Propagate signals, simulate RTL logic

We modeled the RTL logic as various combinational gates connected in a graph. We implemented the ability to simulate these components, updating the values on the wires (edges) as the inputs propagated to the outputs. This laid the foundation to build GLIFT models on top of the combinational elements of the design.

- Implement models for conservative and precise models in combinational logic

With the groundwork done for simulating and following values through the system, we could add the conservative and precise models to the simulated components, thereby constructively building the shadow logic as part of each component in the design.

- Algorithm to print the tracked paths through a system (conservative)

We used the models to find where the information propagated through the system under the conservative model. This was to find and list the potential paths through with the secure information could flow.

- Algorithm to print the tracked paths through a system (precise)

The next step was to change some of the dangerous paths found with the conservative model into the more precise model, and simulate a set of random inputs for which the propagation of information could be tracked. This allows us to see exactly where the information flows, and with what probability. We are able to list these paths and mark certain areas or components as potential trojans.

- Model sequential logic

So far, the work has been focused on modeling and tracking information through combinational logic. Combinational logic is almost instantaneous, and any change to an input or component is immediately reflected in the output. Another category of logic circuits is sequential logic. The outputs to a sequential logic circuit depend not only on the inputs, but on the past state of the system as well. This is primarily done with an additional component, called a register or flip-flop, that stores a past value and only propagates a new value when it is signalled. We needed to be able to simulate the behavior of a flip flop to add sequential capability.

- Implement models for sequential logic

In addition to modeling the behavior of a sequential circuit is the accompanying shadow logic. The function of the shadow logic, and of the sequential elements themselves, are depended on the past state. This means that the signals would have to be propagated through repeatedly, in time frames, greatly increasing the complexity and time necessary to simulate the behavior.

- Add CLI (command line interface) to run the tool

The final step is to add an interface for the user of the final tool to interact with the program. The simplest way is with a command line interface through the computer's terminal.

Our original milestones included both the sequential and combinational portions of the project, grouped as a single milestone. As the quarter progressed, The additional complexity of adding sequential elements became clear. In order to complete a working part of the project within the timeframe of this class, we decided to implement the combinational and sequential portions separately, and unify them at in the final program. However, we ultimately decided to only implement the combinational portion at this time. That way, we could fully implement the combinational portion instead of having both portions incomplete.

While the precise model for information tracking is fully implemented for combinational logic, the current tool will only check a small set of predefined values. This is useful for testing, but will need to be changed to handle a set of randomly generated inputs repeatedly to better illustrate the vulnerable paths with precision. This is a small addition.

A command line interface has been created for the tool, primarily for testing and demonstration. The interface still needs to be more general and flexible. The interface will be updated as the precise model for information flow is changed to reflect the amalgamation of multiple random inputs, as mentioned above.

Conclusion:
The area of hardware security is now getting the attention it deserves. Our efforts have so far tried to tackle one of the more common and easy to comprehend mux type trojan. This trojan is always combinational in nature. The other type of trojan the is also very common is the time bomb trojan which is triggered when an internal counter overflows. This type of trojan is always sequential in nature and is far more complicated to detect. For this project we focused of the mux type trojan. Our algorithm currently doesn't scale up very well. It gives good results quickly for small implementations but for large netlists with around 1000 or more gates, it doesn't scale up that well. The speed of our algorithm also depends on the number of inputs. As our main bottleneck is the updating of the frequency map, the number of iterations that we need to run increases exponentially if we are to get meaningful more accurate results with lesser false positives. Another problem that we encountered was that the trojans that we got were dependent on the input values that we feed to our algorithm. We also encountered a lot of false positives. We can solve this problem by running part 3 of our algorithm for multiple iterations each using random input values. We can then define a threshold (similar to d in part 3 of our algorithm) to select the appropriate trojan.

References --
[1] http://cs.ucsb.edu/~sherwood/pubs/TCAD-11-gliftproof.pdf
[2] http://cs.ucsb.edu/~sherwood/pubs/ASPLOS-09-glift.pdf
[3] http://dl.acm.org/citation.cfm?id=1387714
[4] http://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch/