FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis

Adam Waksman Matthew Suozzo Simha Sethumadhavan

Computer Architecture and Security Technologies Lab
Department of Computer Science
Columbia University
New York, NY, USA
{waksman,simha}@cs.columbia.edu
{ms4249}@columbia.edu

ABSTRACT

Hardware design today bears similarities to software design. Often vendors buy and integrate code acquired from third-party organizations into their designs, especially in embedded/system-on-chip designs. Currently, there is no way to determine if third-party designs have built-in backdoors that can compromise security after deployment.

The key observation we use to approach this problem is that hardware backdoors incorporate logic that is nearly-unused, *i.e.* stealthy. The wires used in stealthy backdoor circuits almost never influence the outputs of those circuits. Typically, they do so only when triggered using external inputs from an attacker. In this paper, we present FANCI, a tool that flags suspicious wires, in a design, which have the potential to be malicious. FANCI uses scalable, approximate, boolean functional analysis to detect these wires.

Our examination of the TrustHub hardware backdoor benchmark suite shows that FANCI is able to flag all suspicious paths in the benchmarks that are associated with backdoors. Unlike prior work in the area, FANCI is not hindered by incomplete test suite coverage and thus is able to operate in practice without false negatives. Furthermore, FANCI reports low false positive rates: less than 1% of wires are reported as suspicious in most cases. All TrustHub designs were analyzed in a day or less. We also analyze a backdoor-free out-of-order microprocessor core to demonstrate applicability beyond benchmarks.

Categories and Subject Descriptors

B.6.2 [**Hardware**]: Logic Design—*Security and Trust*

Keywords

hardware; security; backdoors; functional analysis; intellectual property

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 $15.00.

1. INTRODUCTION

Malicious backdoors and intentional security flaws in hardware designs pose a significant threat to trusted computing [1, 2, 3]. This threat is growing in seriousness due to the ever-increasing complexity of hardware designs. A designer can hide a backdoor within a hardware design by writing one or a few lines of code in a way that slightly deviates from specification. For instance, a hardware backdoor, when triggered, might turn off the page protections for a certain range of addresses or weaken the cryptographic strength of a psuedo-random number generator. Such backdoors can be inserted either by third-party designers producing independent components or by malicious insiders working for an otherwise benign company. As a concrete example, King *et al.* designed a backdoor that triggers when a specific rare value appears on the memory bus [4].

In recent years, techniques have been proposed for protecting against hardware design backdoors, including unused circuit identification [5], validation of design properties at runtime [6], and methods for disabling backdoor triggers at runtime [7, 8]. Each of these solutions provides protection against some of the hardware backdoor attack space, and each of these techniques operates at least partially at runtime. Runtime techniques increase design complexity, due to the added effort of modifying designs to include runtime protections.

A key difference between our work and prior works is that our solution does not depend directly on validation and verification. This is extremely useful because validation and verification teams are often large (larger even than design teams) and hard to trust. Additionally, it can be hard in practice to verify third-party IP.

We propose a solution for discovering backdoors in hardware designs prior to fabrication using functional analysis. If backdoors can be detected statically, then the design can be fixed or rejected before it is taped-out and sent to market. The key insight behind our work – one that has been observed in prior works [5, 7] – is that backdoors are nearly always dormant (by design) and thus rely on *nearly-unused logic,* by which we mean logic that almost never determines the values of output wires. It is desirable to design backdoors with rare triggers to avoid unintentional exposure during design validation or other benign testing. In other words, triggers give stealth and control to the adversary. Our goal is to *statically* identify what we refer to as *weakly-affecting inputs*, which are input wires that have the capability to serve as backdoor triggers.

We propose a metric called *control value* to identify nearly-unused logic. This metric measures the degree of control that an input has on the operation and outputs of a digital circuit. The gist of our method is to approximate the truth table for each intermediate output in a design as a function of any wire that can determine that output. We then compute the influence of each input on the output. We show that control value computations can be approximated efficiently and accurately for real circuits and that control value is a useful measure for finding backdoors. We then present a tool called FANCI—Functional Analysis for Nearly-unused Circuit Identification. FANCI reads in a hardware design and flags a set of wires that appear suspicious. FANCI whitelists most of the design (usually more than 99%) and flags a few suspect wires to be code reviewed.

The intuition behind why FANCI works is that in a given design module, there are typically very few (or even zero) wires with low enough control values to be capable of serving as a backdoor trigger. Typically, a backdoor has more or less the following form: a good circuit and a malicious circuit exist. The outputs of both feed into something semantically equivalent to a multiplexer. The multiplexer is controlled by an input that selects the output of the malicious circuit when triggered. For this general arrangement to work, the control value for the control wire is made very low, and FANCI detects such wires.

While we are not theoretically guaranteed to find all backdoors, our empirical results support that the types of circuits designers create in the real world can be effectively analyzed using our tool. For the backdoored circuits in the TrustHub benchmark suite, we were able to detect all backdoors with low false positive rates. We were also able to analyze a backdoor-free, out-of-order microprocessor core without obtaining false positives, indicating that FANCI does not flag most commonly used circuits as backdoors. We argue that applying FANCI to designs statically prior to applying runtime protections can only bolster defenses and never weakens them. Lastly, our method has pseudo-randomness built in to defeat adversaries that may have knowledge of our tools.

The rest of the paper is organized as follows. We first present our threat model in Section 2. We then describe our analysis algorithm and the way FANCI works in Section 3. We further show that our algorithm solves a previously unsolved problem in backdoor detection in Section 4. The results of our experiments are presented in Section 5. Finally, we discuss related work in Section 6 and conclusions in Section 7.

2. THREAT MODEL

In our threat model, an independent hardware designer or third-party intellectual property (IP) provider supplies us with a hardware design. This design is soft IP, encoded as either hardware description language (HDL, also sometimes referred to as RTL) source code, a gatelist or a netlist. Gatelists are produced by logic synthesis, and netlists are produced by physical synthesis. In each case, the design is a soft product that has not yet been sent to foundries for physical manufacture. The provider is malicious and has included hidden, malicious functionality that they are able to turn on at an opportune time for them. The nature of the malicious payload of the attack is not restricted.

Our goal as security engineers is to use non-runtime, validation- and verification-independent, functional analysis to identify which wires in a digital design could potentially be carrying backdoor signals. We want to flag a small number of wires and be assured that the malicious functionality is dependent on a subset of those suspicious wires. In other words, we need to avoid false negatives (a false negative would mean a backdoor that we do not detect).

False positives are also relevant. We must flag a small enough set of wires that security engineers or code reviewers can evaluate all of the suspicious wires by inspecting code. In other words, we must whitelist most of the design. We consider a wire to be a true positive if it is part of combinational logic used in the triggering of the backdoor. In other words, a true positive wire is part of the circuit that produces malicious activity. Our goal in this paper is detection, not correction. By detecting a backdoor prior to fabrication and deployment, we at least know that our provider is malicious before we apply compromised IP to our designs. We can then blacklist that provider and get our IP from a different source. We do not attempt automatic correction.

3. THE FANCI ALGORITHM AND TOOL

Algorithm 1 Flag Suspicious Wires in a Design

```
for all modules $m$ do
  for all gates $g$ in $m$ do
    for all output wires $w$ of $g$ do
      $T \leftarrow$ TruthTable(FanInTree($w$))
      $V \leftarrow$ Empty vector of control values
      for all columns $c$ in $T$ do
        Compute control of $c$ (Section 3.2)
        Add control($c$) to vector $V$
      end for
      Compute heuristics for $V$ (Section 3.3)
      Denote $w$ as suspicious or not suspicious
    end for
  end for
end for
```

We begin with a high-level overview of the algorithmic steps in FANCI and then describe each step individually. Algorithm 1 describes how suspicious wires are flagged within an untrusted design. For each module and for each gate in the module, we examine the outputs. When we refer to outputs we mean any wire that is the output of any gate, not just the output pins of a chip or a module in the design. Since we are looking at all wires (including internal ones), we do not unnecessarily bias our search for backdoor activity.

For each output wire, we construct a functional truth table for the corresponding inputs (*i.e.* the cone of input wires that feed into the given intermediate output, also called the fan-in tree). We then iterate through each of the input columns of the truth table [1]. For each column, we hold all other columns fixed. For each possible row, we check to see if the value of the column in question determines the output.

Mathematically, there are two different logical functions, the function one gets from fixing the input to digital zero and the function one gets from fixing the input to digital one. We are computing the boolean difference between these two functions. As a result, for each input wire, we get a number between zero and one (inclusive) that represents the fraction of the rows that are influenced or controlled based on the input column. Once we have done this for each input, we have a vector of these numbers. We then apply heuristics (described in Section 3.3) to these vectors to decide if the output wire is suspicious.

All of our analysis is done on a per-module basis. While a hardware backdoor can affect security of operations that happen in multiple modules, *i.e.* the payload of the backdoor can be spread across different modules, the actual trigger computation usually happens in only one circuit within one module. The choice to analyze per-module is practical but not mathematically necessary. As an added benefit, each module can be analyzed independently of each other, which means in the future our tool could be parallelized for improved scalability.

[1] This is similar to what is called zero-delay combinational logic simulation in CAD tools

Table 1: A small example of an unaffecting input dependency. Input C has no influence over the output O.

Input A	Input B	**Input C**	Output O
1	1	1	0
1	1	0	0
1	0	1	1
1	0	0	1
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0

Table 2: An example of an always-affecting input dependency. C influences the value of the output O in every row.

Input A	Input B	**Input C**	Output O
1	1	1	1
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

Before getting into the further details of the algorithm and implementation, we provide some background and relevant terminology regarding digital wires and circuits.

3.1 Terminology

FANCI operates at the level of wires and gates, which are the basic building blocks of digital hardware. In this section we define the notion of dependency, control values and other relevant concepts in terms of these building blocks for the understanding of the FANCI tool and underlying algorithm.

Our goal is to identify suspicious circuits, and a suspicious circuit is one that is nearly unused. In prior related work [5], suspicious circuitry was defined as circuits that are not used or otherwise activated during design verification tests. Our definition contrasts in two key ways. First, we do not care about verification tests. Second, we consider wires to be suspicious if they are activated rarely rather than never at all. In other words, we are not simply looking for unused logic. We are looking for logic that is used rarely or in situations that have a low probability of being exercised during regular testing.

Dependence: We distinguish between two distinct dependence relations that can exist between wires. These are *physical dependence* and *value dependence*. A wire w_2 is *physically dependent* on another wire w_1 if the signal in w_2 receives signal from the wire w_1. In other words, there is a path of combinational logic connecting w_1 to w_2. Thus, the value that w_2 carries comes from computation that makes use of the value carried by w_1. We can also think of w_2 as the output of a small circuit for which w_1 is an input. Outputs are dependent on inputs. When we say that w_2 is dependent on w_1, we refer to physical dependency. If w_2 is dependent on w_1, then we say w_1 is a dependency of w_2. Thus, dependent and dependency are dual notions. A is a dependent of B when B is a dependency of A.

Value dependence means that there is functional dependence. A wire w_2 is value dependent on w_1 if the digital value taken on by w_2 changes depending on the value of w_1. Given that w_2 is physically dependent on w_1, w_1 potentially determines the value of w_2, but it is not guaranteed. For example, in the case of a circuit that always outputs digital one, the input values do not affect the output at all.

We break down value dependence into three relevant types. These terms convey the notion of how much one wire affects or influences another. They are *unaffecting, always-affecting and weakly-affecting*.

Unaffecting Dependency: A dependency of a wire is unaffecting if it never determines the value of its dependent wire. An example of this is shown in Table 1. The truth table shown represents a small circuit with three inputs (A, B, C) and one output (O). There are eight possible cases, broken into four pairings. Within each pairing, the value of C can be zero or one. However, that choice does not matter, because the value of the output O is fixed with respect to C.

The truth table is equivalent to a circuit where O is equal to the logical XOR of A and B. Within each pair of rows, the set of values for O is either all ones or all zeros. Thus, we say that the input C is an unaffecting dependency of the dependent output wire O.

Always-Affecting Dependency: The opposite of unaffecting is always-affecting. A dependency of a wire is always-affecting if the value of that dependency always influences the value of its dependent wire. An example is shown in Table 2. The circuit being represented is similar to the one from Table 1. In this case, however, every pair of rows is affected by the value of the input C.

The truth table is equivalent to a circuit where the output O is computed as the logical XOR of all three of its inputs. In this case, no matter what the values of A and B are, the value of C determines the computed value of the output.

Weakly-Affecting Dependency: Weakly-affecting dependencies are the ones we care about the most in this paper. This is because malicious backdoor triggers rely on weakly-affecting input dependencies for the implementation of nearly useless logic.

A weakly-affecting dependency is a case where one input wire affects an output but only very rarely. One example of this could be a large comparator. Consider a circuit that compares a 64-bit array of wires against the value 0xcafebeefcafebeef. Consider one of those 64 input wires, say the least significant bit. The output wire takes on the value one only if all 64 input wires match the specified comparison value. This means that the least significant bit only matters if the 63 other bits already match. In that case, the least significant bit would make the difference between 0xcafebeefcafebeee and 0xcafebeefcafebeef and thus the difference between and output of zero or one. However, in the other $2^{63} - 1$ cases, that least significant bit is irrelevant. For example, it does not matter if the input is 0xaaaaaaaaaaaaaaaa or 0xaaaaaaaaaaaaaaab. Thus, out of the 2^{63} total input case pairs, there is only a single one in which that input bit matters. Thus, it is a weakly-affecting dependency for the output wire.

In general, a wire w_2 has a weakly-affecting dependency w_1 if in nearly all cases the value of w_1 does not determine the value of w_2. In other words, for some threshold value $\epsilon > 0$ such that $\epsilon \ll 1$, the control value of w_1 on w_2 is less than ϵ.

If we consider the example from Section 1, of a backdoor where a comparator on the memory bus fires for one unique large data value on the bus, all of the input wires to that comparator are clear examples of weakly-affecting dependencies for the output wire that serves as the backdoor trigger.

Algorithm 2 Compute Control Value

```
 1: $count \leftarrow 0$
 2: $c \leftarrow$ Column($w_1$)
 3: $T \leftarrow$ TruthTable($w_2$)
 4: for all Rows $r$ in $T$ do
 5:     $x_0 \leftarrow$ Value of $w_2$ for $c = 0$
 6:     $x_1 \leftarrow$ Value of $w_2$ for $c = 1$
 7:     if $x_0 \neq x_1$ then
 8:         $count$++
 9:     end if
10: end for
11: $result \leftarrow \frac{count}{size(T)}$
```

3.2 Computing Control Values

In this section, we discuss how to compute control values for the dependencies of wires that are the outputs of circuits. The discussion thus far has motivated why weakly-affecting dependencies are stealthy wires of interest. They are necessary for the implementation of malicious backdoors. In other words, if the output of a circuit or gate is carrying a stealthy, malicious signal, then some or all of its inputs are weakly-affecting. We compute control value to quantify how weak or strong the degree of effect is.

Roughly speaking, the control value of an input w_1 on an output w_2 quantifies how much the truth table representing the computation of w_2 is influenced by the column corresponding to w_1. Specifically, the control value is a number between zero and one quantifying what fraction of the rows in the truth table for a circuit are directly influenced by w_1. Note that this is independent of particular tests inputs that might be supplied during validation. Even with high quality test suites, most tests fail to exercise all of the internal circuits because input coverage and code coverage are not equivalent to internal state or value coverage. This provides attackers with an obvious way to hide their backdoors. By operating statically and looking at the truth tables, we can observe the behaviors of every gate in the design.

The algorithm to compute the control value of w_1 on w_2 is presented as Algorithm 2. We note that in step 3, we do not actually construct the exponentially large truth table. We instead construct the corresponding function, which is equivalent to a BDD.

There is one further and necessary optimization we make. Since the sizes of truth tables grow exponentially (with respect to the number of input wires), computing control values deterministically is exponentially hard. Thus, in our evaluation we approximate control values by only evaluating a constant-sized subset of the rows in the truth table. We choose the subset of rows uniformly at random at runtime to make it impossible for attackers to know which rows we will choose. This algorithm is depicted in Algorithm 3.

To take a simple example, suppose we have a wire w_2 that is dependent on an input wire w_1. Let w_2 have n other dependencies. From the set of possible values for those n wires (2^n-many), we choose a constant number, let us say for instance 10,000. Then for those 10,000 cases, we set w_1 to zero and then to one. For each of the 10,000 cases, we see if changing w_1 changes the value of w_2. If w_2 changes m times, then the approximate control value of w_1 on w_2 is $\frac{m}{10{,}000}$.

The fact that we choose the inputs at random is important. Backdoors can be designed to evade known validation test suites. Only by choosing at random can we guarantee that the attacker will not know what part of the truth table is going to be explored. Our hypothesis, which is supported by our results in Section 5, is that choosing a constant, large number of inputs at random is sufficient for the weak law of large numbers to take effect, resulting in small statistical deviations and high quality approximations.

Algorithm 3 Compute Approximate Control Value

```
$numSamples \leftarrow N$ (usually $2^{15}$)
$n \leftarrow$ number of inputs
$rowFraction \leftarrow \frac{numSamples}{2^n}$
$count \leftarrow 0$
$c \leftarrow$ Column($w_1$)
$T \leftarrow$ TruthTable($w_2$)
for all Rows $r$ in $T$ do
    if $rand() < rowFraction$ then
        $x_0 \leftarrow$ Value of $w_2$ for $c = 0$
        $x_1 \leftarrow$ Value of $w_2$ for $c = 1$
        if $x_0 \neq x_1$ then
            $count$++
        end if
    end if
end for
$result \leftarrow \frac{count}{numSamples}$
```

3.3 Heuristics for Identifying Backdoors from Control Values

When we are finished computing approximate control values for each input, we have a vector of values for each output of each gate in the design. In this section we describe the heuristics that we use for making final decisions about wires in designs. Given a vector of control values, these heuristics determine whether or not a wire is suspicious enough to be flagged for inspection. For example, having only one weakly-affecting wire or a wire that is only borderline weakly-affecting might not be sufficiently suspicious. This might be a wire that is in the same module as a backdoor but has no relation to it. Or it could simply be a benign but slightly inefficient circuit. This is why we need heuristics for taking into account all of the control values in the vector.

Going back to the example where w_2 is our output, w_2 has a vector of $n+1$ control values from its inputs (w_1 and the n others), each between zero and one. These $n+1$ numbers are the $n+1$ control values from the dependencies of w_2. In this section, we discuss options for processing these vectors to make a distinction between suspicious and non-suspicious output wires.

For a small but real example of what these vectors can look like, consider a standard, backdoor-free multiplexer with two selection bits that are used to select between four data inputs. This common circuit is depicted in Figure 1. The output M of the multiplexer is dependent on all four data inputs and both selection bits. Semantically, the selection bits choose which of the four data values is consumed.

We can see intuitively what the control values are for the six input wires (computation for one input is shown explicitly in Figure 1). The situation is symmetric for each of the four data wires (A, B, C and D). They directly control the output M in the cases when the selection bits are set appropriately. This occurs in one fourth of the cases, and each of these data inputs has control value 0.25. This can also be confirmed by writing out the truth table and counting the rows.

The two selection bits have higher control values. A given selection bit chooses between two of the data values. For example, if $S_1 = 1$ then S_2 chooses between B and D. In that case S_2 matters if and only if $B \neq D$, which occurs in half of the cases. So the control values for the two selection bits are 0.50. The full vector of

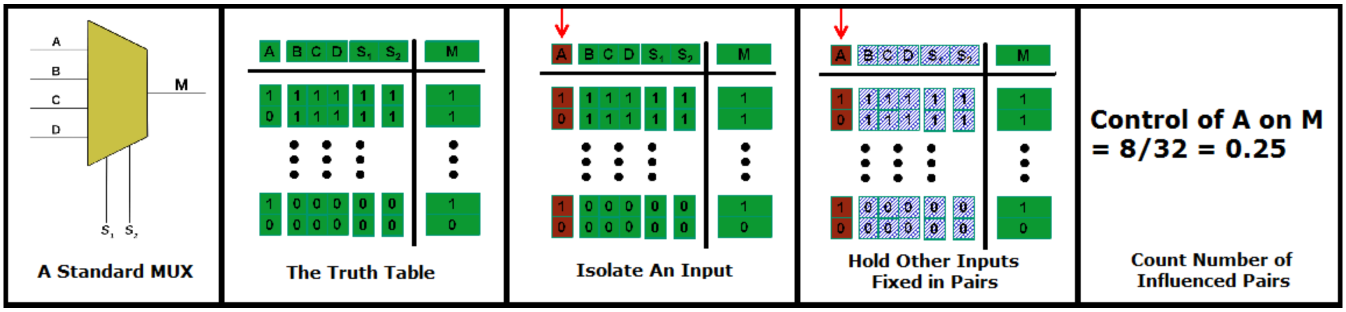
A
B
C
D
M
S1 S2
A Standard MUX
A B C D S1 S2 M
The Truth Table
Isolate An Input
Hold Other Inputs Fixed in Pairs
Control of A on M = 8/32 = 0.25
Count Number of Influenced Pairs

Figure 1: A standard 4-to-1 multiplexer. The output M takes on the value of one of the four data inputs (A, B, C, D) depending on the values of the two selection bits (S_1, S_2).

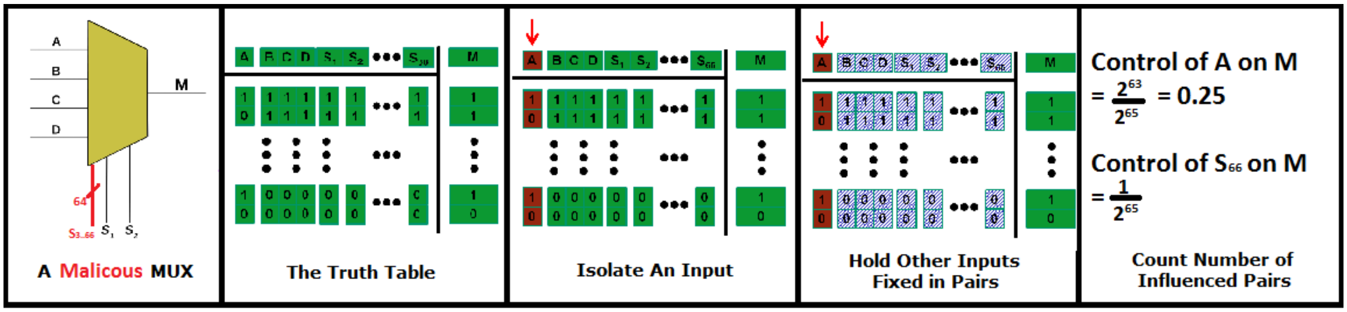
A
B
C
D
M
64
S3..66 S1 S2
A Malicous MUX
A B C D S1 S2 ••• S66 M
The Truth Table
Isolate An Input
Hold Other Inputs Fixed in Pairs
Control of A on M = 2^63 / 2^65 = 0.25
Control of S66 on M = 1 / 2^65
Count Number of Influenced Pairs

Figure 2: A malicious 4-to-1 multiplexer. The output M takes on the value of one of the four data inputs (A, B, C, D) depending on the values of the two selection bits (S_1, S_2). There are also 64 extra selection bits ($\{S_3, \cdots S_{66}\}$) that only change the output if they match a specific key.

control values for the output M contains six values, one for each of the six inputs. The values are:

$$[0.25, 0.25, 0.25, 0.25, 0.50, 0.50]$$

Intuitively, this is a benign circuit, as we would expect. All of the inputs are in the middle of the spectrum (not close to zero and not close to one) which is indicative of a common and efficient circuit.

Figure 2 depicts a malicious version of a multiplexer. There are 64 additional select bits. When those 64 bits match a specific 64-bit key, then the output of the multiplexer is changed to a malicious payload. In terms of the truth table, this affects only an exponentially small fraction of the output rows. The vector of control values we would get for the output M would include 64 additional values for those 64 extra input wires. Each of those control values would be on the order of 2^{-63}. Intuitively, this is an suspicious circuit, as it is a textbook backdoor. We next discuss heuristics for interpreting these vectors.

From a large circuit or large design, we get a variety of these control value vectors, one each per intermediate output within the circuit. The guarantee we have about the distribution of control values is at least one or a few of them will be zero or nearly zero for wires that belong to stealthy backdoor triggers. Thus, the vectors will contain at least some small values. The practical question is how to deal with these vectors and identify the output wires that are truly suspect from ones that are benign. Toward this end, we consider a few (but not all) different heuristics for evaluating these vectors. The general description is shown in Algorithm 4.

Median: The first option we consider is the median. In the case of backdoor triggers, the wires on the critical paths of the trigger generally have mostly unaffecting or very weakly-affecting dependencies, such as in the example displayed in Figure 1. Thus, the median is often close to zero. The median can be an imperfect metric when the data distribution is irregular, which does happen. Using only the median (as we confirm in our evaluation), can result in a few unnecessary false positives.

Algorithm 4 Compute a Heuristic for an Output Wire

```
$w \leftarrow$ output wire
$h \leftarrow$ heuristic function (e.g., median)
$t \leftarrow$ threshold (between zero and one)
$v(w) \leftarrow$ vector of control values
$result(w) \leftarrow h(v(w))$
if $result(w) < t$ then
    return suspicious
else $\{result(w) \geq t\}$
    return not suspicious
end if
```

Mean: In addition to the median, we also consider the mean of the control values. The mean is slightly more sensitive to outliers. For example, if there are only a few dependencies, and one of them is unaffecting, that is likely to get noticed.

Both: Since there are potential limitations with both median and mean, we also consider the option of using both, *i.e.* flagging wires that have extreme values for both the mean and the median. We set a threshold for both the median and the mean, and we flag a wire as suspicious only if the median value is low and the mean value is also low. This helps in some cases to slightly diminish false positives. Details and comparisons are presented in Section 5.

Triviality: One last heuristic we consider in our implementation is one that we call *triviality*. In terms of control values, this is a weighted average of the values in the vector. We weight them by how often they are the only wire influencing the output to determine how much an output is influenced overall by its inputs. The purpose is to learn more about the output wire and less about the individual inputs. For instance, in a circuit that XORs many inputs, each input has a control value of 1.0, but it is never the case that one input completely controls the output, and the triviality of the output is only 0.5. Equivalently, this heuristic computes the fraction of the rows in the truth table in which the dependent output wire has the value zero (or symmetrically, the value one). In practice, we can compute triviality directly in this way by looking only at the output column, which often allows for faster runtime than the other heuristics.

The name 'triviality' refers to the fact that if the triviality value is zero or one then the circuit is completely trivial (always outputs zero or always outputs one). This metric quantifies how functionally trivial the sub-circuit computing a specific output wire is. Note that this metric is not a simple function of the control values, as it makes use of correlations, but we went with it because it worked well in practice. The exact value for triviality can vary from run to run depending on which rows are randomly selected, but it is probabilistically likely to vary by only a very small amount. Empirically, we did not see significant variance. Additionally, since triviality can be computed in this alternative way, it might be a good metric for unusually large modules or if computational runtime becomes relevant.

For each metric, it is necessary to have a cut-off threshold for what is suspicious and what is not. This value (between zero and one) can be chosen either *a priori* or after looking at the distrubtion of computed values. In practice, the latter often works better, as there are often natural breakpoints to choose as the threshold. Either way, the threshold is generally very small, *i.e.* $\ll .001$. Various other heuristics and/or thresholds could be considered in the future to attempt to gain improvements in terms of false positive rates.

4. RELATION TO STEALTHY, MALICIOUS CIRCUITS

Prior to our work, UCI [5] was the state-of-the-art in analyzing backdoors inserted during the design phase. The state-of-the-art in design backdoor attacks is a class of attacks known as *stealthy, malicious circuits* (SMCs) [9]. This class of attacks deterministically evades UCI and was a viable way to attack hardware designs prior to our work. As we will see, FANCI catches SMCs with high probability (approaching 1).

UCI is an analysis algorithm that looks at dataflow dependencies in hardware designs and looks for completely unused intermediate logic. It is a form of dynamic validation; in terms of our terminology, they identify dependencies that are always-affecting dependencies for a given test suite. Given the inputs in the test suite, if two wires always carry the same values as each other, there is an identity relationship, and the internal logic is unneeded. If the test suites were exhaustive, then UCI would have significantly fewer false positives. However, given the incompleteness of standard validation test suites, UCI has many false positives. For this reason, the Bluechip system was built to replace the removed logic with exception handlers that invoke runtime simulation software whenever false positives are encountered.

There are a few key differences between FANCI and UCI. The first is that FANCI does not require a validation test suite. This is valuable for two reasons. Today, third-party IP blocks often do not come with a validation test suite. Furthermore, if a validation suite is supplied, the malicious provider can change the validation test suite to help the compromised hardware evade UCI. A common problem in validation and verification is that achieving good code coverage and good interface coverage does not mean good coverage of internal states and wires. Certain rare states may never get tested at all, which can lead to bugs in commercial designs and also offers ways for backdoor designers to evade detection, such as misusing 'don't care' states. FANCI tests all logic equally, regardless of whether or not it is an input interface, and so it is impossible for a portion of the logic to go untested.

The second key difference between UCI and FANCI is that UCI is deterministic and discrete-valued in its approach. Given a test suite, a wire is only flagged if it is completely unused, regardless of its relations to other wires. In FANCI, we also catch nearly-unused wires, meaning wires that are not completely unused but which rarely alter output signals. For example, if a wire strongly affects the value of a nearby wire (and thus is not quiescent) but ultimately has only a small impact on an output wire a few hops away, we will notice that. A wire that is part of a backdoor trigger might also do useful work in a different part of the circuit, and we account for that. Another aspect of FANCI is that it takes into account the full vector of dependencies and uses heuristics to make a final decision. For example, if a wire affects two different outputs, one in a reasonable way and one only rarely, FANCI can notice that. In the designs we tested, there were many always-affecting dependency relationships that FANCI correctly did not flag. Those relationships could have been false positives in UCI.

To give a toy example, consider a double-inverter path, two inverters placed one after the other. This is a logical identify function, so it generates an always-affecting relationship that would be flagged by UCI. However, as long as the output of the double-inverter path is used, it would not be flagged by any of FANCI's current heuristics. This is a small example and could easily be hard-coded for in a practical implementation of UCI. However, it serves as a microcosm of the difference between the deterministic approach of UCI and the heuristic-based approach of FANCI.

Sturton *et al.* introduced stealthy, malicious circuits as a way to evade UCI. FANCI detects SMCs, and we explain the intuition behind why that is. The basic idea behind SMCs is to use logic that alters the values of intermediate wires but ultimately does not affect outputs. Using this backdoor class, Sturton et al. demonstrated basic circuit building blocks — such as AND and OR gates — that can be used to implement stealthy hardware backdoors. Thus, any small backdoor can be turned into an SMC and evade UCI. The truth table for one of the simplest SMCs is the following (reproduced from [9]):

t_1	t_0	i_1	i_0	h	f	Operation
0	0	0	0	0	0	Normal Operation
0	0	0	1	1	0	Normal Operation
0	0	1	0	0	0	Normal Operation
0	0	1	1	1	1	Normal Operation
0	1	0	0	0	0	Normal Operation
0	1	0	1	1	0	Normal Operation
0	1	1	0	0	0	Normal Operation
0	1	1	1	1	1	Normal Operation
1	0	0	0	0	0	Normal Operation
1	0	0	1	1	0	Normal Operation
1	0	1	0	0	0	Normal Operation
1	0	1	1	1	1	Normal Operation
1	1	0	0	1	1	Malicious Operation
1	1	0	1	1	0	Malicious Operation
1	1	1	0	1	1	Malicious Operation
1	1	1	1	1	1	Malicious Operation

There are two normal input bits i_1 and i_0 and two trigger bits t_1 and t_0. In terms of the output f, this is a classic backdoor trigger. Only when all of the trigger bits are set to one does the functionality change. In the other cases, the functionality is fixed, and the circuit looks like f is the AND of i_1 and i_0. The use of the intermediate variable h, which is distinct from f, makes it so that t_1 and t_0 are not truly quiescent. Thus, Sturton proved that UCI's defenses could be evaded.

Can FANCI detect stealthy, malicious circuits? Observe that the trigger wires – t_1 and t_0 – are weakly-affecting for the output f, *i.e.,* they only affect the value of f during malicious operation, which is a smaller fraction compared to normal operation. This fraction diminishes as the number of trigger bits increases. Thus, for the backdoors in this class of stealthy, malicious circuits, the trigger inputs will have low control values and will be caught by FANCI with high probability.

5. EVALUATION

For our implementation of FANCI, we developed a parser for gatelists that are compiled from the Verilog HDL, a popular language for hardware design. The concepts and algorithms we apply could be applied to VHDL or any other common HDL, as well as to hand-written gatelists. Though our analysis is language agnostic, we use Verilog for all evaluation purposes. We use benchmarks from the TrustHub suite, a popular benchmark suite for work on hardware backdoors [10]. TrustHub is a suite from an online community of hardware security researchers and includes a variety of different types of backdoors, intended to be both stealthy and effective. For some of these benchmarks, the gatelists were provided. For others, we acquired the gatelists from the Verilog source using the Synopsys logic synthesis tool, DC Compiler.

From a given gatelist, our goal is to construct a circuit representation that can be used to calculate different types of dependencies. We treat multiple-bit wires as sets of independent wires. Gates that represent multiple basic logic functions — such as an AND-OR-INVERTER (AOI) — are treated as functionally equivalent to their basic elements. We treat memory elements (*e.g.,* flip-flops) as their logical equivalents. For example, a D-flip-flop is treated as an identity function. We do this because exponential state-space exploration is infeasible, and as such treating state machines as stateful, rather than as their combinational counterparts, would be impractical. Since we track all internal wires (as opposed to only inputs and outputs), we catch sequential backdoors by catching the combinational logic used during internal state recognition.

5.1 Results for Detecting Backdoors

We evaluate the four heuristics presented in Section 3.3 on the TrustHub benchmarks. We perform one run on each design[2] with $2^{15} = 32,768$ input cases (truth table row pairs), with the row pairs chosen uniformly at random (without replacement).

The most important result is that we did not encounter false negatives. For each benchmark and each of the heuristics, we discovered at least one suspicious wire from each backdoor, which was enough for us to identify the functionality of the hidden backdoors. Different metrics can highlight different parts of the backdoor. The mean and median tend to highlight backdoor payload wires and are more similar to each other than to triviality. We hypothesize that this is because these payloads have triggers or resulting values from triggers as their inputs. Thus, several of the input wires have low control values. On the other hand, triviality focuses more on the output wire itself and as such tends to highlight backdoor trigger wires. Since these are wires that rarely toggle, their truth tables tend to score very low for triviality. Using multiple metrics in concert can help out in code review by flagging more of the wires associated with the backdoor and thus demarcating the boundary of the backdoor more clearly.

Figure 3 shows the results for the 18 TrustHub benchmarks we analyzed regarding false positives. For our results, we categorize the benchmarks into groups as they are categorized by TrustHub. These categories represent four different design types, containing a variety of backdoor triggering mechanisms. Each of the four groups contains a variety of backdoors manually included into a given design. The RS232 group contains eleven benchmarks, representing eleven different backdoors applied to a relatively small third-party UART controller. The S35932 and S38417 groups each contain three benchmarks, containing backdoors built into two gatelists whose source and description are not provided. The S15850 group contains only one benchmark. The S38417 group contains the largest designs in terms of area and number of gates, while the RS232 benchmarks, as the smallest, mostly contain sequential triggers. The s15850, s35932, and s38417 categories are qualitatively different from RS232 and more similar to each other. We experienced a decrease in false positive percentage for these larger designs, which we attribute to the fact that the total number of false positives did not vary significantly with respect to design size.

Additionally, the different benchmark categories achieve differing degrees of stealth (some are documented and others can be calculated manually). The stealth is imply the probability that a backdoor will accidentally reveal itself on a random test input. Most of the triggers in the RS232 category have a relatively high probability (*i.e.* low stealth) of going off randomly, as high as around one in a million. In the other categories, the probabilities are lower, ranging from one in several million to as low as around one in 2^{150}. The backdoors in the three low probability groups are the most realistic, since they are stealthy enough to evade detection by normal methods. The backdoors in the RS232 category go off with such high probability that validation testing would have a good chance of finding them. This is an aspect that made them more difficult to distinguish and resulted in slightly more false positives. From what we have empirically observed, the larger the design and the more well-hidden the backdoor, the better FANCI performs in terms of keeping false positive rates low.

Unsurprisingly (as shown in Figure 3), using the median by itself produced the most false positives on average. However, the difference is not large. The heuristic that produced the least false positives on average was triviality. All four metrics are effective enough for practical use. We also believe that other metrics could be considered in the future to achieve incremental improvements. A promising result we discovered was that the percentage of false positives diminished as we looked at larger designs (granted this is a small sample set). In other words, it appears that scaling up to larger designs does not greatly increase the total number of false positives (*i.e.* the effort of code review).

Figure 4 shows how many wires are flagged as suspicious on average for each of the benchmark groups by each of the different metrics. Each of the four metrics worked well, though the mean turned up the most suspicious wires on average (at the cost of slightly higher false positive rates). We see that all four metrics flag only a small number of critical wires, which means security engineers are given a small and targeted set to inspect. For most of the benchmarks, FANCI whitelists more than 99% of the designs, making code review and inspection a feasible task.

[2] If desirable, multiple runs could be performed to increase confidence. In practice, the same results tend to come up every time, but it cannot hurt.

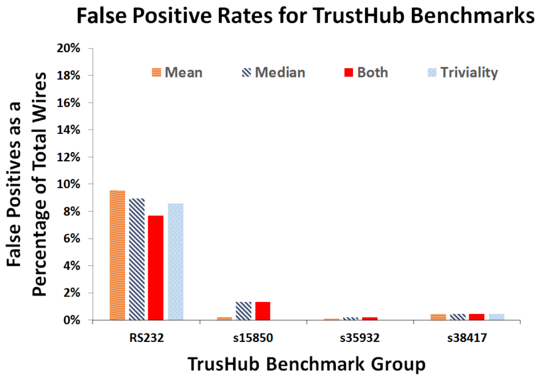

Figure 3: False positive rates for the four different metrics and for TrustHub benchmarks. The RS232 group — which is the smallest — has about 8% false positives. The others have much lower rates (less than 1%).

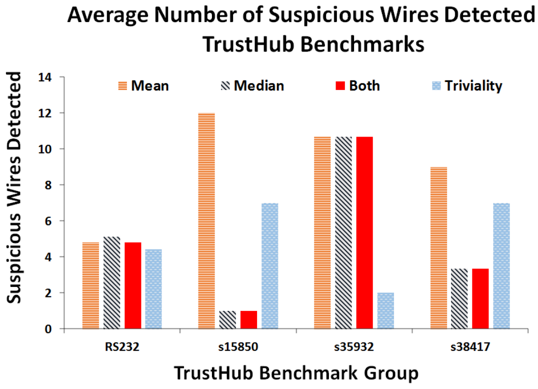

Figure 4: These are the total number of suspicious wires detected by each method for each type of backdoor design on average. For each design and each of the four methods we tried, we always found at least one suspicious wire. Thus, each of the four methods is empirically effective. However, some turned up larger portions of the trigger critical paths, proving to be more thorough for those cases.

We lastly test to see what happens as we increase and decrease the number of input rows we sample. The results are shown in Figure 5. We see that up to a certain point, the results improve. After that point, the results tend to converge and stay roughly the same. This is essentially the weak law of large numbers kicking in, and it allows FANCI to scale well. Note that due to randomness, sometimes we flag more values using less inputs. This ends up not affecting our results significantly, since the true positives tend to be clustered in the design, so adding or removing one wire does not make a large difference in code review. What we also learned from varying the number of inputs is that there are two sources of false positives. The first source is approximation. If we run only a few inputs, we get extra false positives, and if we run more inputs we get less false positives. The second source is from persistent positives, i.e. weakly-affecting signals that are in the design for legitimate reasons. The first type disappears quickly as the number of inputs gets large, which is why false positives due to approximation are not a major concern.

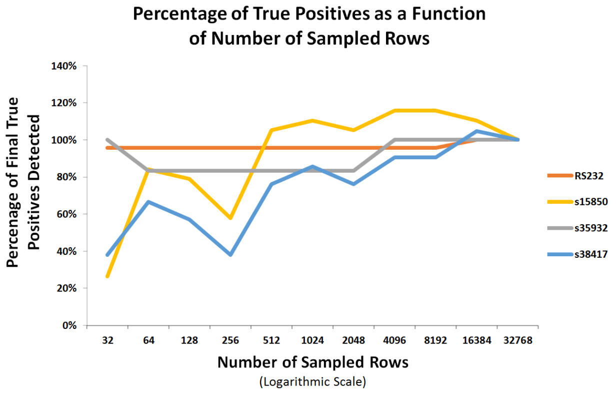

Figure 5: The trade-off between the number of inputs being used (*i.e.* running time) and the percentage of true positives caught, normalized to the results for 2^{15} inputs. Results are shown averaged over the four different metrics we used. The x-axis is on a logarithmic scale.

5.2 Runtime and Random Row Selection

The runtime for FANCI is roughly proportional to the size of the design under test in terms of number of total gates. In practice, the runtime for a normal module ranges from less than an hour to a couple of days using 2^{15} row pairs per approximate truth table. The runtime can be increased or decreased by changing the number of inputs tested. In practice we did not find it necessary to decrease this number. Given the sizes of third-party IP components on the market, the runtime for FANCI should not be a major problem for real-world scenarios. Our runtime in terms of number of gates scales similarly to many synthesis and analysis tools, since our tool and other tools require the parsing of every gate in the design.

To be precise, the asymptotic runtime for a deterministic algorithm would be in the set $\mathcal{O}(nd2^d)$ where n is the number of gates (or nodes) and d is the maximal degree of a node, *i.e.* the maximal number of inputs on which an intermediate wire can depend. Using approximate truth tables reduces the asymptotic runtime to the set $\mathcal{O}(nd)$. Making the (usually true) assumption that the degree is small and bounded, this reduces to the set $\mathcal{O}(n)$, which represents linear runtime. The algorithm is trivially parallelizable, since the algorithm is in essence a massive *for* loop. Our initial implementation is sequential, but in the future it could be made parallel if necessary.

Lastly, we do not do directed testing or targeting of specific rows in truth tables or specific inputs. We go with uniform randomness because any other method would be better for an attacker and worse for us as the security engineers (assuming the attacker knows our strategy).

5.3 Discussion of False Positives

One lesson learned from our experiments is that false positives tend to be consistent and are not greatly affected by the randomness of our sampling methods. We anticipate that the false positives we encounter in the future will bear similarities to each other, perhaps allowing for easier recognition. Some examples of potential false positives could be the most significant bit of a large counter or an input to an exception-recognition circuit. These circuits are semantically similar to backdoors, because they react to one specific rare case. For example, consider a floating point divider that throws a single exception, caused by a divide-by-zero error. Then for the data input representing the divisor, only the value zero invokes the exception-handling logic. The exception-handling logic is nearly-unused.

The existence of these circuits should not pose much of a problem, because counters and exceptions are easily recognizable in code review. Nevertheless, as an attacker, one could be motivated to include many such circuits to increase the false positive count. The problem from an attacker's point of view is that each of these false positives requires a costly circuit, and so building an entire design this way would be impractical. Additionally, these types of circuits tend to have obvious architectural purposes, and so adding thousands of them would be a dead giveaway in code review. For example, including a large number of exception handlers that serve no apparent purpose would be a source of concern during code inspection.

Our hypothesis was that in real designs (*i.e.* designs that one might buy as commercial IP), even malicious designers are forced to follow common design conventions and design reasonably efficient circuits. We believe that this is the reason we did not find a significant number of false positives in any of the designs we analyzed.

A related and important property of our approach is that it behaves well with respect to common, reusable structures. In modern designs, much of the circuitry is spent on reusable components, such as CAMs, RAMs, FIFOs, decoders, encoders, adders, registers, etc. For some simple designs, such as adders and multipliers, the results of FANCI have been mathematically verified. We have not had issues with false positives for these common types of structures. When identifying suspicious wires, we look for outliers. In these standard structures, there tend to be no outliers due to symmetry. Consider a CAM with 32-bit data entries. For each entry, there is a 32-bit data comparator, which includes some very low control value dependencies (on the order of $\frac{1}{2^{32}}$). However, each of the comparators is identical (or nearly identical), leaving no outliers to serve as false positives. Additionally, the nature of the structure should make it obvious in code review how many such wires should exist (often a power of two or otherwise documented number).

5.4 Out-of-Order Processor Case Study

In order to study FANCI on a larger and backdoor-free design, we use the FabScalar microprocessor core generation tool [11]. FabScalar is an HDL code generator that produces processor cores given a set of parameters. The core we choose to use is a moderately-sized, out-of-order core with four execution units and does not contain backdoors.

The core we analyze has a total of 56 modules. The modules contain about 1900 distinct wires on average, with the largest module containing slightly over 39,000 distinct wires. This largest one is abnormally large for a single module containing primarily combinational logic. However, as this is an auto-generated design, it is understandable. If it were being hand-written, it most likely would be broken into smaller, coherent pieces. While the overall design is larger than any of the modules from the TrustHub suite, and larger than typical third-party IP components, many of the individual modules are on average around the same size as modules in the TrustHub suite.

We were able to analyze each of the 56 modules in FabScalar using 2^{15} row pair samples per truth table, except for two of the abnormally large modules where we had to approximate more coarsely. The two largest modules are outliers and took several days to process, even using more coarse-grained approximation. These could more easily be analyzed in a commercial setting on a compute cluster. Additionally, many software optimizations (including parallelization) could be applied prior to commercialization.

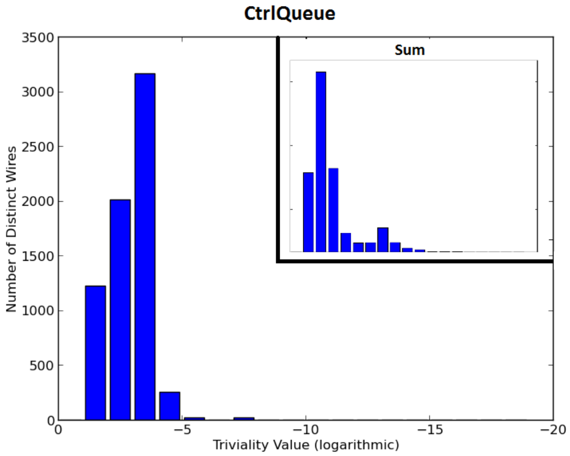

Figure 6: A histogram of the triviality values for wires in a typical FabScalar module called CtrlQueue. The biggest spikes occur at around $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$, which is common. There are no major outliers. X-axis values are shown on a logarithmic scale, starting at one and getting smaller going to the right. Inlaid in the upper right is the sum of all 56 FabScalar modules.

As expected, we did not detect false positives in the benign FabScalar core. To garner further intuition for how our heuristics look for wires in benign hardware, we construct a histogram of a typical FabScalar module (shown in Figure 6). In this example, there are two major spikes at $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$. The reason for the presence of spikes is that semantically similar wires tend to have similar values, as we saw in the example of a multiplexer. For this module, there are no suspicious outliers, with all of the values being more than 0.01 (and less than 0.99). We did not see any noticeable outliers, and our thresholds are typically less than 0.001. More data from FabScalar is included in the Appendix.

5.5 Security Discussion and Limitations

We briefly discuss some of the security properties discussed in this paper and the corresponding limitations.

- FANCI relies on the assumption that backdoors use weakly-affecting wires. This is valid in practice because they need to be stealthy. The more well-hidden the backdoor is, the more likely it is to be caught by FANCI because more well-hidden backdoors have lower control values. It is provable[3] that for a fixed-length combinational circuit path, achieving a given level of stealth requires a correspondingly low control value for one or more of the inputs. On the other hand, the less well-hidden it is, the more likely it is to evade FANCI but be caught during testing. We would call such an attack a ***Frequent-Action Backdoor***, where the idea is to put the backdoor in plain sight. Standard validation testing and FANCI are highly complementary.
- FANCI does not remove the need for standard code inspection/review practices. Consider as an example an attack where a malicious designer includes hundreds of backdoor-like circuits. Each of these circuits could turn on given a variety of rare triggers, with only one of them having a useful malicious payload. Thus, FANCI would flag all of them, mostly generating false positives. We would call this type of attack ***False Positive Flooding***. However, in addition to

[3] We leave out the full proof as it is out of scope for this venue.

the area bloat this would cause, it would be obvious in basic code inspection that this was not a reasonable design. FANCI specifically targets small, well-hidden backdoors, which are the type that are able to evade testing and code inspection.

• Functional analysis only applies to designs or discrete representations of designs. Functional analysis alone does not protect against backdoors inserted physically into a device by a malicious foundry, unless a functional representation can be reverse engineered from the device via decapping, which is not easy. We would call these types of attacks ***Physical or Parametric Backdoors.*** Functional analysis is one piece of hardware security and must exist as part of the larger security scope, which includes validation, code inspection and foundry-level techniques in addition to runtime methods

Our approach also works well against sequential backdoors but with limitations. Sequential backdoors are trigged not by one combinational input but by a stream of small inputs over time. In other words, they are triggered by the combination of an input and a specific internal state. Hypothetically, a sequential backdoor that makes use of an extremely large and contrivedly deep state machine might be able to evade detection or at least made detection more difficult. We would call an attack of this type a ***Pathological Pipeline Backdoor***. The idea is that by stretching out the backdoor trigger computation over a long stretch of logic, it makes the control value data more noisy and potentially more difficult to interpret. For example, if an input needs to determine an ultimate output with $\frac{1}{2^{32}}$ probability, this can be done with two sequential components, each with a probability of $\frac{1}{2^{16}}$ of turning on. The control value from beginning to end will still be $\frac{1}{2^{32}}$, but there will be many intermediate control values, and the overall metrics might not be as clean.

This scenario is one of the many cases where we find that FANCI is complementary to standard validation practices. While basic tests would be likely to catch an extremely large backdoor, FANCI is more likely to catch small, well-hidden backdoors. As we can see in Table 3, practical backdoors tend to have relatively small critical path lengths, and none of the backdoors we have encountered have used deep pipelining. In the table, we use path length (in number of gates) as a proxy for the depth and size of a backdoor trigger computation. These results could be interpreted as merely commentary on the specific types of backdoors that benchmark designers choose to build, or they could be interpreted as broadly representative of the way attackers build malicious circuits. Without a wider array of benchmarks, we cannot say for certain. However, it appears that the crucial part of a backdoor – even a relatively complex backdoor – tends to be composed of only a few gates, and this is good for security engineers.

Table 3: Average Length of Backdoor Critical Paths in TrustHub Benchmarks

TrustHub Benchmark Group	Average Backdoor Path Length
RS232	4.9
s15850	5.0
s35932	4.4
s38417	4.0

6. RELATED WORK

Hardware design backdoor detection, identification, categorization and protection are areas that have recently grown in interest. Hardware designs have been demonstrated to be highly vulnerable [1, 4]. Reese *et al.* evaluated how lightweight and stealthy one can make practical backdoors [12]. In recent years, there has been work both in design-time and in-the-field or runtime protection schemes.

Hicks *et al.* proposed a runtime method for averting backdoors [5]. This method has been shown to detect backdoors, thus raising the bar for the cleverness of hardware attacks. However, it is also vulnerable to sophisticated attacks, as demonstrated by Sturton *et al.* [9] and discussed in Section 4. The three key differences between our work and theirs: 1) our detection technique is exclusively design-time, 2) we do not rely on a validation suite to identify suspicious circuits, and 3) we provide a continuous measure of suspiciousness as opposed to a binary metric used by Hicks *et al.*

Also in the area of runtime techniques, Waksman and Sethumadhavan designed *TrustNet* [6], a methodology for integrating property checkers into hardware designs that can ensure that a wide array of properties are upheld at runtime. Waksman and Sethumadhavan also developed a technique for disabling digital hardware backdoors at runtime, which identifies possible trigger sources and prevents backdoor triggers from reaching malicious logic [7, 8]. Their work identifies the notion of a trigger as a rare signal that does not fire during validation testing. Our work with FANCI is complementary to that prior work, in part because it lessens the burden of trust on validation teams.

There has also been prior work in related areas of hardware supply-chain security, including the detection of physical backdoors added during fabrication [13, 14, 15, 16] and detecting actively running backdoors [17, 18]. This work generally assumes a trusted design, called a golden model, which we and others endeavor to make more of a reality.

The concept of influence of input variables over truth tables and boolean functions has been approached from a theoretical perspective at least as far back as 1988 [19]. As far as we know, we are the first to apply these concepts to hardware security. Our work does not rely on formal verification or require manual inspection or understanding of the inner-workings of designs.

7. CONCLUSIONS

The ability to identify and understand hardware backdoors at design time using static analysis mitigates the dangers of integrating third-party intellectual property components into hardware. We presented a concept called *control value*, which describes how wires within a design affect other wires. Using the idea of control value, we developed a methodology for identifying suspicious wires that have the capability to carry backdoor trigger signals. Specifically, we look at the influence wires have over intermediate outputs within a circuit and identify those wires that have an abnormally low degree of influence. Our method is scalable and approximate; to achieve our goals, we build truth tables of intermediate outputs in the circuit of interest and compute the control value by randomly sampling rows in the truth table. Using a tool we developed, called FANCI, we examined 18 TrustHub benchmarks. We were able to identify triggers in each of these benchmarks, obtaining low false positives rates (flagging less than 10 wires per design on average) in the process.

FANCI is the first tool for checking the security of third-party soft IP and regular hardware designs prior to fabrication. Similar to software static analysis tools, we envision FANCI being used as

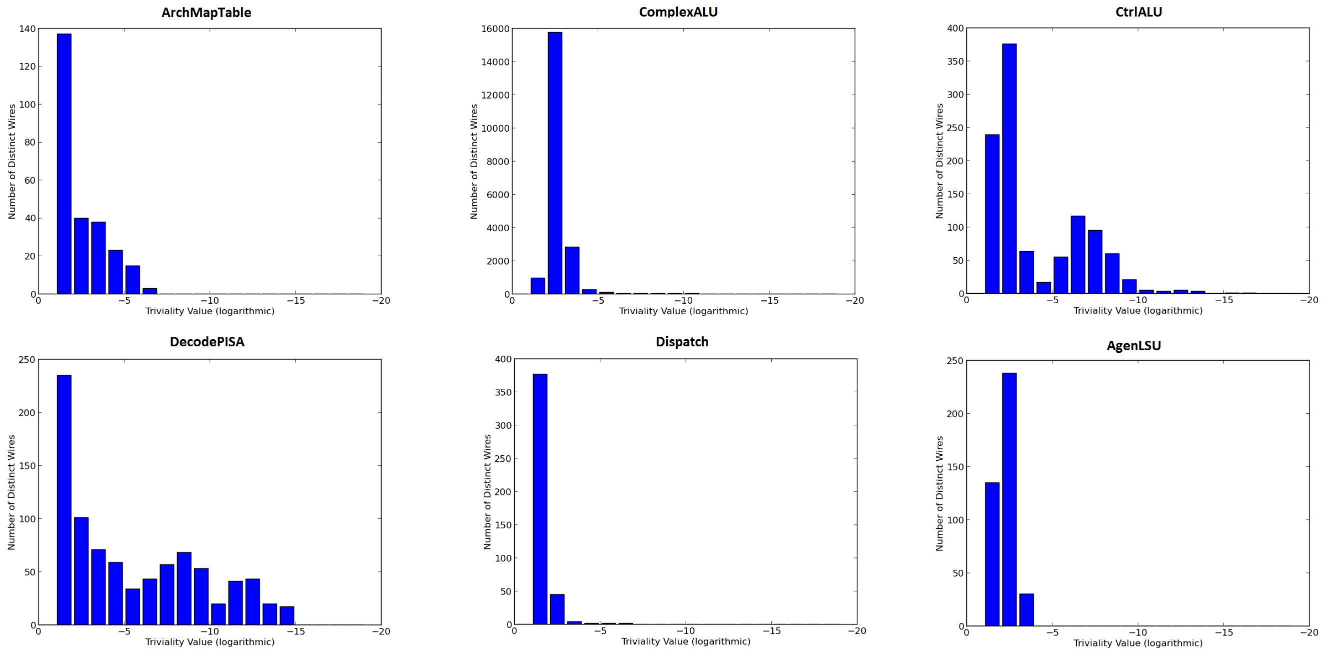

Figure 7: Histograms of the triviality values from three of the modules in the FabScalar core design we used. The y-axis shows the number of wires in each category, and the x-axis shows a logarithmic scale of the triviality values for the wires. Triviality scales from one to zero (going left to right), so the logarithmic values scale from zero to negative infinity.

a first line of defense for enhancing hardware security. It is complementary to runtime techniques for protecting against hardware attacks and also to standard testing practices. Additionally, it has fewer trust requirements as compared with previously existing runtime detection/protection techniques. While our tool is not theoretically guaranteed to find all backdoors, it is likely that backdoors that evade FANCI have to break the digital abstraction or have to be non-stealthy and thus detectable through normal means. Our experimental results support the claim that this methodology could be applied to real-world designs today. As designs get more complex and time to market shrinks, tools like FANCI that can target backdoors prior to fabrication are critical to the development of trustworthy systems.

ACKNOWLEDGEMENTS

We thank anonymous reviewers and members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback on this work. This work was supported by grants FA 99500910389 (AFOSR), FA 865011C7190 (DARPA), FA 87501020253 (DARPA), CCF/TC 1054844 (NSF) and gifts from Microsoft Research, WindRiver Corp, Xilinx and Synopsys Inc. This work is also supported through an Alfred P. Sloan Foundation Fellowship and the Department of Defense NDSEG Fellowship. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US Government or commercial entities.

APPENDIX

In Figure 7, we include some example histograms of the triviality values we found for wires in six of the modules from FabScalar, the benign microprocessor core that we tested with FANCI. In a normal design, most of the wires have values that are not extremely small, with values between $\frac{1}{8}$ and $\frac{1}{2}$ being very common. To make the results easier to read, we have combined the values between zero and $\frac{1}{2}$ with the values between $\frac{1}{2}$ and one. For example, 0.1 and 0.9 are plotted together, as are 0.7 and 0.3. Semantically, we care about the distance from $\frac{1}{2}$, so this is the easiest way to understand the data.

To take the example of the DecodePISA module, which experienced slightly lower triviality values than the other example modules, it turns out that most of the lower values belong to higher order bits of a 128-bit output packet called DecodedPacket0. Without knowing the intention of the original designer, it seems likely that these upper order bits are not always being used efficiently. However, the control values are not so low as to merit real suspicion. In addition to serving as a security method, these types of observations may also be useful for regular debugging and optimization by trusted designers.

As we can see in the histograms, the vast majority of wires are bunched up on the left side, having relatively normal values (closer to $\frac{1}{2}$ then to the extremes of zero or one). In FabScalar, we rarely see wires with values even less than 2^{-10}, which is still a relatively benign value (corresponding to roughly a one in one thousand chance of a certain behavior occurring). We can also see that while the values are mostly close to $2^{-1} = \frac{1}{2}$, the actual distributions vary from module to module. This is to be expected, as module designs are complex, and it is rare for two different modules to be exactly the same.

References

[1] Sally Adee. The Hunt for the Kill Switch. *IEEE Spectrum Magazine*, 45(5):34–39, 2008.

[2] Marianne Swanson, Nadya Bartol, and Rama Moorthy. Piloting Supply Chain Risk Management Practices for Federal Information Systems. In *National Institute of Standards and Technology*, page 1, 2010.

[3] United Stated Department of Defense. *High Performance Microchip Supply*, February 2005.

[4] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 5:1–5:8, Berkeley, CA, USA, 2008. USENIX Association.

[5] Matthew Hicks, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 159–172, 2010.

[6] Adam Waksman and Simha Sethumadhavan. Tamper Evident Microprocessors. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 173–188, Oakland, California, 2010.

[7] Adam Waksman and Simha Sethumadhavan. Silencing Hardware Backdoors. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 49–63, Oakland, California, 2011.

[8] Adam Waksman, Julianna Eum, and Simha Sethumadhavan. Practical, Lightweight Secure Inclusion of Third-Party Intellectual Property. In *Design and Test, IEEE*, pages 8–16, 2013.

[9] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 64–77, Washington, DC, USA, 2011. IEEE Computer Society.

[10] Mohammad Tehranipoor, Ramesh Karri, Farinaz Koushanfar, and Miodrag Potkonjak. TrustHub. http://trust-hub.org.

[11] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg. Fabscalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 11–22. IEEE, 2011.

[12] Trey Reece, Daniel Limbrick, Xiaowen Wang, Bradley Kiddie, and William Robinson. Stealth Assessment of Hardware Trojans in a Microcontroller. In *Proceedings of the 2012 International Conference on Computer Design*, pages 139–142, 2012.

[13] Sheng Wei, Kai Li, Farinaz Koushanfar, and Miodrag Potkonjak. Provably Complete Hardware Trojan Detection Using Test Point Insertion. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '12, pages 569–576, New York, NY, USA, 2012. ACM.

[14] Dakshi Agrawal, Selçuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan Detection using IC Fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 296–310, 2007.

[15] Mainak Banga, Maheshwar Chandrasekar, Lei Fang, and Michael S. Hsiao. Guided Test Generation for Isolation and Detection of Embedded Trojans in ICS. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 363–366, New York, NY, USA, 2008. ACM.

[16] Jie Li and J. Lach. At-Speed Delay Characterization for IC Authentication and Trojan Horse Detection. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 8–14, June 2008.

[17] Mainak Banga and Michael S. Hsiao. A Region Based Approach for the Identification of Hardware Trojans. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 40–47, June 2008.

[18] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. New Design Strategy for Improving Hardware Trojan Detection and Reducing Trojan Activation Time. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pages 66 –73, 2009.

[19] Jeff Kahn, Gil Kalai, and Nathan Linial. The Influence of Variables on Boolean Functions (Extended Abstract). pages 68–80, 1988.