Defeating UCI: Building Stealthy and Malicious Hardware

Cynthia Sturton
University of California, Berkeley

Matthew Hicks
University of Illinois, Urbana-Champaign

David Wagner
University of California, Berkeley

Samuel T. King
University of Illinois, Urbana-Champaign

***Abstract*—In previous work Hicks et al. proposed a method called Unused Circuit Identification (UCI) for detecting malicious backdoors hidden in circuits at design time. The UCI algorithm essentially looks for portions of the circuit that go unused during design-time testing and flags them as potentially malicious. In this paper we construct circuits that have malicious behavior, but that would evade detection by the UCI algorithm and still pass design-time test cases. To enable our search for such circuits, we define one class of malicious circuits and perform a bounded exhaustive enumeration of all circuits in that class. Our approach is simple and straight forward, yet it proves to be effective at finding circuits that can thwart UCI. We use the results of our search to construct a practical attack on an open-source processor. Our malicious backdoor allows any user-level program running on the processor to enter supervisor mode through the use of a secret "knock." We close with a discussion on what we see as a major challenge facing any future design-time malicious hardware detection scheme: identifying a sufficient class of malicious circuits to defend against.**

***Keywords*-hardware; security; attack**

I. Introduction

A computer's hardware layer is often trusted implicitly. As a result, a machine that contains untrustworthy hardware may be open to attack, regardless of its operating system or software stack. If the hardware contains a backdoor, an attacker may be able to gain total control of the machine, bypassing any security protections provided by the software. This is true even if the OS and application layers are free of bugs and vulnerabilities. Recent research has demonstrated how just such an attack could work [1], [2], and the media have started to report on the United States government's concern about the possibility of these attacks occurring in the real world [3], [4].

There has been a variety of research on the problem of detecting malicious hardware (see Section VII for an overview). In this paper we focus on the Hicks et al. paper [2] which tackles the problem of detecting malicious hardware inserted at design time. The authors of that work propose an algorithm, Unused Circuit Identification (UCI), for identifying portions of a circuit that go unused during design-time testing. The idea is that an attacker inserting malicious code into an existing hardware design will work to ensure the malicious behavior is not activated during design-time testing. The assumption is that any circuitry inserted by

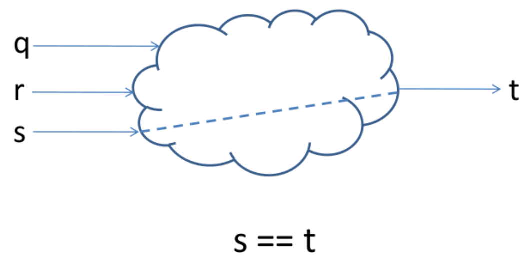


Figure 1. *The UCI Algorithm.* Suppose s carries the same value as t for every test case executed during design-time testing. This means that the intervening logic could have been replaced by a single wire, without invalidating any of the tests. In this case, the UCI algorithm flags the circuitry between s and t as potentially malicious.

the attacker will remain inactive during the entirety of the design-time testing process.

The UCI algorithm works by creating a dataflow graph corresponding to the source code under test and then looking for any pair of signals (s, t), such that t is dependent on s and the intervening logic between s and t could have been replaced by a wire without affecting any of the outputs computed during design-time testing. If such a pair is found, that pair identifies a portion of the circuit that does not appear to have been activated during design-time testing and thus might be malicious. Figure 1 illustrates the idea behind UCI.

In this work, we identify circuits that have hidden (i.e., malicious) behavior, yet can evade detection by the UCI algorithm. In particular, these circuits have the following property: for all dependent signal pairs (s, t), there exists at least one configuration of inputs that would result in $s \neq t$ and would not cause the circuit to exhibit the hidden behavior. This configuration of inputs, if present in the suite of test cases used during design-time testing, ensures the UCI algorithm will not flag the circuitry between s and t as potentially malicious. In other words, in our circuits there is no pair of dependent signals that are always equal during design-time testing.

To produce these attack circuits, we developed a search algorithm for finding circuits that are malicious (have hidden behavior), admissible (would pass design-time testing), and stealthy (would evade UCI). The algorithm is straightforward, a bounded exhaustive enumeration of possible combinational logic circuits from a given set of gates, yet it proves powerful in finding hardware that defeats UCI.

Using one of the circuits returned by our search, we implement a practical attack against an open-source processor. We insert a backdoor into an open-source processor and show that the backdoor is *not* detected by the UCI algorithm. Our backdoor allows a user-level program that knows the secret handshake to enter supervisor mode and thus take control of the system. The behavior of our attack is identical to one described in earlier work on malicious backdoors in hardware [2], but unlike prior work, our backdoor is constructed in a way that evades detection by UCI. Our attack targets the Leon3 processor,[1] an implementation of the SPARCv8 architecture.[2] We synthesize a Leon3 processor with our backdoor added and show that a user-level program running on Linux on the processor can exploit the attack and cause the processor to transition into supervisor mode by executing a special sequence of instructions.

After describing our practical attack, we look at ways UCI might be strengthened against the type of attacks we built. Our work suggests there is no easy fix to UCI, and indeed, any UCI-like algorithm that depends solely on test cases (which are necessarily non-exhaustive) for the specification of correct behavior will always run into difficulty. We identify what we see as a major challenge for any future UCI-like algorithm: identifying an adequate class of malicious circuits to defend against. We discuss these topics further in Section VI.

A. Contributions

This research contributes to the field of malicious hardware detection in the following ways:

- We show that UCI, the malicious hardware detection scheme proposed by Hicks et al. [2], is flawed. We design and implement an attack on the Leon3 processor that can be exploited to launch a software-level attack. The malicious hardware evades detection by UCI, while still allowing the processor to pass design-time testing.
- We present an approach for finding malicious circuits that can be easily tailored and replicated for evaluating future research in the area of malicious hardware detection. We use this approach to find simple circuits that contain malicious behavior, but that UCI would not be able to detect.
- We present what we feel is a major challenge facing any future design-time malicious hardware detection scheme: identifying an adequate class of malicious circuits to defend against.

[1]http://www.gaisler.com

[2]http://www.sparc.org

B. Threat Model and Assumptions

We follow the same threat model and assumptions introduced by Hicks et al. [2]. Namely, we assume a rogue designer can insert malicious hardware into the circuit design, but the attack needs to remain hidden during traditional design-time testing. The rogue designer cannot control the suite of tests used for design-time testing of the hardware, but can learn arbitrary information about the test cases (e.g., which instructions will appear in a test case).

II. Background: UCI

Before describing our approach to finding attacks on UCI, we first explain how the UCI algorithm works. The following description is intended only to give a high level understanding of how the algorithm works. The technical details can be found in the original paper [2].

UCI operates on the design of some circuit, given in some hardware description language (HDL).[3] The algorithm proceeds in four phases: static analysis, instrumentation, verification, and classification. In the first phase, a dataflow graph is constructed from the source code (which may or may not contain hidden malicious logic). In the graph, each edge represents a signal (wire) and each node represents a gate. The dataflow graph is used to find the set of all dataflow pairs: the set of pairs of signals (s, t) where there is some path from s to t in the dataflow graph. Such a path indicates that data can flow from signal s to signal t. Another way to think about it is that signal t is dependent on signal s.

Once the set of dataflow pairs has been identified, the original source code is modified to include tracking hardware: signals that note whenever any dataflow pair becomes unequal, i.e., whenever $s \neq t$.

The third phase of the algorithm is verification. The newly instrumented code is simulated and run through a battery of tests. At the end of the verification phase, the tracking signals show which, if any, dataflow pairs were equal throughout all test cases. These dataflow pairs indicate places where malicious circuitry may lie. If there is a dataflow pair (s, t) for which the property $s = t$ holds throughout all test cases, then it follows that the circuitry between s and t could be replaced with a single wire (i.e., short-circuited) without affecting the result of any test case. This intermediate circuitry is highlighted as potentially malicious, as this logic seems to serve no purpose in any of the test cases.

III. Constructing Attacks: Methodology

In this section, we define a class of circuits that can evade detection by UCI and that can be used for malicious

[3]Hicks et al. [2] state that their approach would work equally well on a netlist describing the circuit. However, their exposition assumes the HDL is provided and therefore, in our description, we do as well. This issue has no bearing on the ability of UCI to detect our attacks.

purposes. Then, we conduct a systematic search for circuits in this class, with the following approach:

1) We fix an upper bound on the number of input signals to the circuit and on the size of the circuit (total number of gates). Also, we fix the type of gates that may be used in the circuit.
2) We enumerate all circuits of the class which fall within the bounds.
3) We analyze each circuit generated and keep those that evade UCI, meet our definition of malicious, and could still pass design-time testing.

All circuits produced by this search defeat UCI and could be used by an attacker to insert a stealthy, malicious backdoor into a target hardware design. We define search criteria that are sufficient for a circuit to evade detection by UCI, but the criteria are by no means necessary: there may be other ways to construct circuits that defeat UCI. The search criteria are designed to keep the search practical. As shown in Section IV, the class of circuits the search produces is rich enough to defeat UCI and produce a malicious circuit that can be used in a practical attack.

A. *Definitions*

Before defining the class of circuits targeted by our search, we define the following terms:

Basis of Gates:

The basis of gates is a set $G = \{g_0, g_1, \ldots, g_n\}$ of logic gates available for use in building circuits. Circuits generated by our search will use only these gates.

Circuit:

For our purposes, a circuit C is some combinational logic built using gates from the basis. $C(x)$ denotes the output of C on input x. We focus on circuits that produce a single bit of output, so a circuit C defines an *output function* f from n-bit inputs to 1-bit outputs: namely, the function $f : \{0,1\}^n \to \{0,1\}$ given by $f(x) = C(x)$. The output function f specifies only the input-output behavior of the circuit, whereas the circuit C itself also records the internal structure of the circuit.

Each internal wire w of C computes some function f_w. In our search algorithm, for each circuit C we store only its output function f and the set $S = \{f_w : w \text{ is an internal wire of } C\}$ of functions for each internal wire of C. For conciseness, we often identify the circuit C with this information: $C = (f, S)$. Given an arbitrarily complex circuit design, we can compute f, S recursively, as follows:

1) If C is a single wire connecting an input x to the output, it computes the projection function $f(x) = x$ and has no internal wires: $C = (f, \emptyset)$.
2) Suppose $C_1 = (f_1, S_1)$ and $C_2 = (f_2, S_2)$ are circuits, and suppose the circuit C is formed by combining the outputs of circuits C_1 and C_2 with a 2-input gate g from the basis of gates, i.e., $C = g(C_1, C_2)$. Then the resulting circuit can be expressed as
$$C = (g(f_1, f_2),\ S_1 \cup S_2 \cup \{f_1, f_2\}).$$
A similar pattern applies to gates with more or less than 2 inputs.

Size:

The size of the circuit is the total number of gates used to construct the circuit.

Trigger Condition:

We define two types of inputs to every circuit constructed: trigger and non-trigger. The non-trigger inputs are the normal inputs, which are used in the normal functionality of the circuit, while the trigger inputs are used only to define a rare condition under which the hidden malicious behavior will be activated. In particular, the hidden behavior should appear if and only if the set of trigger inputs are driven with a single, specific, predetermined, value (the trigger value). The trigger condition is the condition that the trigger inputs are driven with that specific value. Thus, under the trigger condition, a malicious circuit exhibits its special hidden behavior.

The input x to the circuit is decomposed into $x = (i, t)$, where $i = (i_0, \ldots, i_\ell)$ denotes the non-trigger inputs and $t = (t_0, \ldots, t_m)$ denotes the trigger inputs.

Non-Trigger Condition:

A circuit is in the non-trigger condition whenever the trigger inputs are not driven with the trigger values, i.e., when the trigger condition is false. Under the non-trigger condition, the circuit exhibits its expected behavior.

Admissible Circuit:

We say that a circuit is *admissible* if: (1) there exists exactly one trigger condition, (2) the circuit has at least one non-trigger input, and (3) under the non-trigger condition, the trigger inputs become don't-cares (do not have any effect on the output value of the circuit). Our search algorithm only examines admissible circuits.

The admissibility requirement defines a clear separation between trigger and non-trigger inputs and ensures that any constructed malicious circuits will still pass design-time testing.

Note that while the definition of admissibility requires the output of the circuit to be independent of the particular values on the trigger inputs in the non-trigger condition, we do *not* place a similar restriction on non-trigger inputs. In the trigger condition, the output of the circuit is permitted to depend upon non-trigger inputs (the non-trigger inputs do not need to be don't-cares in this case).

Obviously Malicious Circuit:
We define a circuit to be *obviously malicious* if there exists any two valuations to the inputs to the circuit $x = (i, t)$, $x' = (i, t')$ where t is in the non-trigger condition and t' is in the trigger condition and $C(x) \neq C(x')$. In other words, the circuit is malicious if changing just the trigger inputs from a non-trigger value to the trigger value can change the output of the circuit (presumably, activating its hidden functionality).

Stealthy Circuit:
We define a circuit to be *stealthy* if UCI will not flag any part of the circuit as malicious. We assume the test suite used by UCI includes every possible input where the trigger condition is false. (We discuss the implications of this assumption in Section VI.) Thus, a circuit will be stealthy if there is no pair of dependent wires that are always equal during design-time testing. To be more precise, a circuit $C = (f, S)$ will be stealthy if there is no pair s_i, s_j of internal wires ($s_i \in S$, $s_j \in S \cup \{f\}$) such that data can flow from s_i to s_j, and such that $s_i = s_j$ always holds for all inputs satisfying the non-trigger condition.

B. Defining Malice and Requiring Admissibility

Our definitions of admissible and obviously malicious circuits require further discussion. We emphasize that these are intended to serve as a sufficient, but not necessary, condition under which a circuit could be used in an attack. They are by no means an exhaustive characterization of when a circuit might contain a hidden backdoor. For instance, we can imagine circuits that do not have a clear separation between trigger and non-trigger inputs, and which exhibit hidden behavior if the entire input satisfies a certain rare condition; such circuits might be useful to an attacker, but would not be found by our search. Our definitions are intended solely to make algorithmic search easier, by focusing the search on a particular type of dangerous circuit. Our results show that this is sufficient to find serious attacks against the UCI algorithm—but we do not claim that it can be used to create all possible attacks.

Our focus on circuits that have a clean separation between trigger and non-trigger inputs is motivated by the need for some objective way to recognize malicious circuits. Consider, for example, a single AND gate with two inputs and one output. Without any further context, there is no meaningful way to classify this circuit as benign or malicious. Since we can not determine *a priori* whether a constructed circuit is malicious, we instead look for hidden functionality. Hidden functionality should be triggered only in a special condition that is unlikely to arise in normal design-time testing. A simple special case arises when some of the inputs are used only to trigger the hidden functionality, and do not otherwise affect the circuit's behavior. This motivates separating the inputs into non-trigger vs. trigger inputs. If the circuit exhibits new behavior when the trigger inputs are set to a special value, then we consider that the circuit has hidden behavior and thus may be malicious.

We impose the admissibility requirement only to facilitate algorithmic search. It is possible to imagine a malicious circuit in which all inputs affect the output at all times, but one particular combination of inputs produces an unwanted, malicious behavior. (Indeed, in a real-world setting, this is a particularly plausible type of attack.) However, recognizing that such a circuit is malicious requires some semantic knowledge of desirable vs. undesirable behavior.

To see why this is so, we consider an example. Suppose C is a circuit with two trigger inputs $t = (t_0, t_1)$ and with trigger condition $t_0 \wedge t_1$ (i.e., the trigger condition holds if $t = (1, 1)$). The admissibility constraint requires

$$\forall i \ . \ C(i, (0, 0)) = C(i, (1, 0)) = C(i, (0, 1)),$$

i.e., so long as the trigger inputs are not both "1", they do not affect the output of the circuit. If the circuit is admissible, we can easily recognize it as malicious if we have

$$\exists i, t \ . \ C(i, t) \neq C(i, (1, 1)),$$

since this means that when the trigger inputs are fed with a special value (namely, $(1, 1)$), the circuit behaves differently.

Now assume that the circuit C is not admissible. In this case, it would be easy to find a pair of inputs $(i, t), (i, t')$ such that $C(i, t) \neq C(i, t')$, which implies that either $C(i, t) \neq C(i, (1, 1))$ or $C(i, t') \neq C(i, (1, 1))$. This is true regardless of whether the circuit is malicious or not. In other words, if C is not admissible, it is always true that

$$\exists i, t \ . \ C(i, t) \neq C(i, (1, 1)),$$

whether C is malicious or not. Therefore, in the absence of any kind of specification of desirable behavior, there is no clear basis for distinguishing a malicious circuit from a non-malicious circuit.

Even though we only include admissible circuits in our search results, as we show in Section IV-B, we were able to easily translate one of the circuits from our search results into a real-world attack.

C. Methodology

With the definitions in place, we can now explain our methodology more precisely. We enumerate all circuits in the class defined by the following parameters:

- 1 or 2 non-trigger inputs: $i = (i_0, i_1)$ or $i = (i_0)$.
- 2 trigger inputs: $t = (t_0, t_1)$.

- Basis of gates given by the set G, where $G \subseteq \{\text{AND}, \text{OR}, \text{NOT}, \text{NAND}, \text{2-input MUX}\}$.
- Trigger condition given by $t_0 \wedge t_1$ (i.e., the trigger condition holds if $t = (1, 1)$).
- Circuit size: $N \leq 3$.

For each circuit enumerated, if it is admissible, obviously malicious, and stealthy, we add it to our set of malicious circuits that can evade detection by UCI.

D. Searching for Circuits

Our search algorithm works as follows. We maintain a workqueue of newly formed circuits—these will serve as building blocks for future circuits. We also maintain a set of completed circuits R. Roughly speaking, in each iteration of the algorithm we remove one circuit C from the workqueue, consider all ways to expand C by adding one gate, add each new circuit generated in this way to the workqueue, and then add C to the list of completed circuits.

More precisely, we initialize the search by setting $R := \emptyset$ and populating the workqueue with four circuits: $C_0 = (i_0, \emptyset)$, $C_1 = (i_1, \emptyset)$, $C_2 = (t_0, \emptyset)$, $C_3 = (t_1, \emptyset)$. In each iteration, we remove one circuit from the front of the workqueue, say $C = (f, S)$. If C is admissible, obviously malicious, and stealthy, we output C and the search continues with the next circuit in the queue. Otherwise, for each gate $g \in G$ and each circuit $C' \in R \cup \{C\}$, we build a new circuit C'' as $C'' = g(C, C')$. If C'' is stealthy and not in R, it is appended to the workqueue, otherwise it is discarded. Once C has been combined with every circuit in $R \cup \{C\}$, using every gate in G, C is added to the set R.

This approach only explores circuits where every subcircuit is stealthy. Note that for a circuit C to be stealthy, it is necessary for every subcircuit of C to be stealthy; therefore, there is no point in exploring circuits with a subcircuit that is not stealthy. Also, the algorithm does not distinguish between two circuits C_1, C_2 with the same output function and set of internal functions: if $C_1 = (f, S)$ and $C_2 = (f, S)$, then these two circuits are indistinguishable in terms of UCI and in terms of their externally observable behavior, so we keep one of C_1, C_2 and discard the other. This partitions circuits into equivalence classes and only keeps one representative of each equivalence class in the workqueue and completed set.

We conducted the above search multiple times, each time using a different basis G of gates. In particular, we conducted a separate search for each subset of $\{\text{AND}, \text{OR}, \text{NOT}, \text{NAND}, \text{2-input MUX}\}$, excluding the empty set. The pseudo-code for the algorithm is shown below. We summarize the results of this search in Section V-D.

Algorithm 1 Searching for stealthy, admissible, obviously malicious circuits

```
// Initial circuits, one for each input.
$C_0 = (i_0, \emptyset)$
$C_1 = (i_1, \emptyset)$
$C_2 = (t_0, \emptyset)$
$C_3 = (t_1, \emptyset)$

// The set of already-considered circuits.
completed_circuits = $\emptyset$
// The circuits to use as building blocks in larger circuits.
workqueue = $[C_0, C_1, C_2, C_3]$
// The set of gates to use in building circuits.
gate_basis = $\{\text{AND}, \text{OR}, \text{NOT}\}$

/* While there are still circuits in the workqueue, pop the next one off and combine it with every previously-considered circuit and with itself. */
while length(workqueue) > 0 do
  curr_circuit = workqueue.pop()
  if curr_circuit is stealthy, admissible, and obviously malicious then
    print curr_circuit
  else
    for all gate in gate_basis do
      for all circ in completed_circuits ∪ {curr_circuit} do
        new_circuit = gate(curr_circuit, circ)
        if new_circuit is stealthy and not in completed_circuits then
          workqueue.append(new_circuit)
        end if
      end for
    end for
    completed_circuits.add(curr_circuit)
  end if
end while
```

IV. Attacks

From the set of admissible, obviously malicious, and stealthy circuits returned by our search, we select a small circuit to analyze in order to gain an understanding of how it evades UCI. We then select a more complex circuit with the desired trigger condition and non-trigger condition output behavior for use in creating a privilege escalation attack in the Leon3 processor. We show that the attack evades UCI analysis.

A. Example Circuit

The smallest circuit we found that satisfied our criteria was a two-gate circuit with basis $G = \{\text{MUX}\}$. This circuit is shown below in Figure 2. The inputs to the circuit are $i = (i_0, i_1)$ (non-trigger inputs) and $t = (t_0, t_1)$ (trigger inputs). The trigger condition is $t_0 \wedge t_1$. The output function of the circuit is $f = (i_0 \wedge i_1) \vee (t_0 \wedge t_1 \wedge \neg i_1)$. Under non-trigger conditions the output function is $f_{NT} = i_0 \wedge i_1$. Under trigger conditions the output is $f_T = i_0 \vee \neg i_1$.

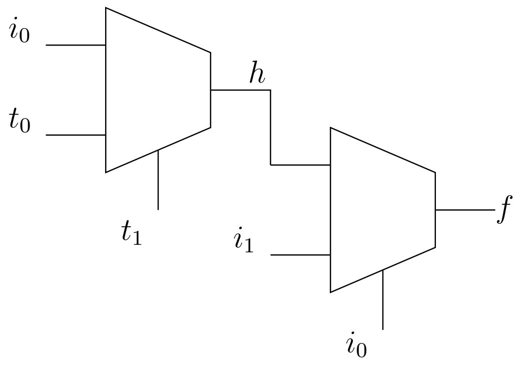


Figure 2. Two-gate, stealthy, admissible, and malicious circuit.

This circuit evades detection by UCI because there is no pair of dependent signals that are always equal under the non-trigger condition. The truth table in Figure 3 shows the input values and associated output values for this stealthy MUX circuit. In addition to showing the output function f, the table shows the values for the lone internal signal h. The rows below the dashed horizontal line are the signal values under the trigger condition, which are not seen during design-time testing. The final column notes when an input makes a dependent pair of signals unequal. The data in the table shows that if the test suite includes the inputs $(t_1, t_0, i_1, i_0) = (0, 0, 0, 1)$, $(0, 1, 0, 1)$, $(1, 0, 0, 1)$, and $(1, 0, 1, 0)$, then UCI will not flag any part of the circuit as malicious. Since the trigger condition for all of these inputs is false and $f_{NT} = i_0 \wedge i_1$, for every test case this circuit will behave identically to an AND gate applied to i_0 and i_1.

If a rogue designer has access to the source code of some security-critical hardware, he can replace any AND gate with the malicious circuit in Figure 2 and cause it to behave differently under a special condition that might not be exercised during design-time testing. If this affects some security-critical element of the hardware, it might introduce a hidden vulnerability or backdoor that is not detected by UCI.

UCI misses this behavior because the way the circuit was constructed, there is no intermediate function equivalent to $i_0 \wedge i_1$ that UCI can short-circuit the output with. This idea of creating a non-trigger function which is *not* equivalent to any internal function is the key to defeating UCI.

B. Practical Attack

The previous example indicates there are ways to construct simple circuits with hidden behavior that evade UCI. In this section, we show how an attacker can replace the functionality of a larger design with one of these circuits

t_1	t_0	i_1	i_0	h	f	Comments
0	0	0	0	0	0	
0	0	0	1	1	0	$h \neq t_1$, $h \neq t_0$, $h \neq i_1$
0	0	1	0	0	0	
0	0	1	1	1	1	
0	1	0	0	0	0	
0	1	0	1	1	0	$f \neq t_0$, $f \neq i_0$, $f \neq h$
0	1	1	0	0	0	
0	1	1	1	1	1	
1	0	0	0	0	0	
1	0	0	1	0	0	$h \neq i_0$
1	0	1	0	0	0	$f \neq t_1$, $f \neq i_1$
1	0	1	1	0	1	
1	1	0	0	1	1	Trigger condition is true.
1	1	0	1	1	0	Trigger condition is true.
1	1	1	0	1	1	Trigger condition is true.
1	1	1	1	1	1	Trigger condition is true.

Figure 3. Truth table for two-gate circuit shown in Figure 2.

containing some hidden behavior and use the hidden behavior to create an attack. We insert our malicious hardware into a processor and show that UCI does *not* detect the attack.

The UCI paper describes what the authors call a *malicious foothold*: malicious hardware inserted into a processor that allows the attacker to later gain entry to the machine and launch a system-level attack. One such foothold they describe is the supervisor transition foothold. This is a piece of hardware inserted into the processor's core pipeline that, upon detecting a secret trigger sequence of instructions ("knock"), transitions the current process into supervisor mode.

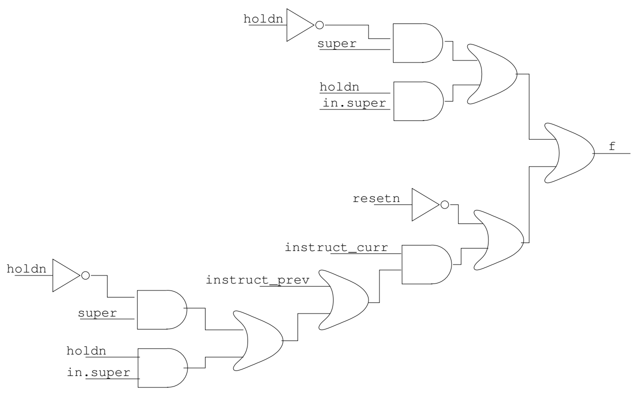
Hicks et al. [2] show that UCI is able to detect their implementation of the supervisor transition foothold. In this section, we show how to implement the same attack, but in a way that UCI can not detect. We introduce a backdoor into the Leon3 processor, which implements the SPARCv8 architecture. The configuration of the processor and system are identical to the configuration used in the paper proposing UCI.

In order to implement a stealthy attack, we looked for a place in the code where the value of the supervisor-mode bit was set by the output of some n-input function `super` $\Leftarrow g(i_0, i_1, \ldots, i_{n-1})$. Then we searched through our library of example stealthy, admissible, and malicious circuits to find one with the appropriate output function. We needed $f_{NT} = g(i_0, i_1, \ldots, i_{n-1})$ and $f_T = 1$, which will, when triggered, artificially increase the privilege level of the executing process.

We inserted our attack in to the processor's integer unit pipeline. The process we modified is one that writes values computed in the previous clock cycle by each stage in the pipeline for use by the subsequent stages in the next clock cycle. Shown in Figure 4 is the fragment of the process pertaining to the setting of the super signal, the bit that will put the processor in supervisor mode. (This is not the actual code, but a more readable representation of the code's behavior.) On `holdn` $= 1$ [4] the supervisor-mode bit gets updated with the value computed in the previous cycle (`in.super`). On `resetn` $= 0$ (active low) the processor is put in supervisor mode. Otherwise, the supervisor-mode bit remains as it was in the previous cycle.

```
reg : process (clk)
  begin
    if rising_edge(clk) then
      if (holdn = '1') then
        super <= in.super;
      end if;
      if resetn = '0' then
        super <= '1';
      end if;
    end if;
end process;
```

Figure 4. Original processor pipeline code responsible for setting the supervisor-mode bit. This is the code before we inserted our backdoor.



```
super <= ((NOT holdn AND super) OR
          (holdn AND in.super)) OR
 ((NOT resetn) OR (instruct_curr AND
     (instruct_prev OR ((NOT holdn AND super)
          OR (holdn AND in.super)))));
```

Figure 5. The attack circuit we use for the hidden backdoor. The top shows the structure of the attack. The bottom shows the HDL implementation of the attack.

Our attack sets `super` $\Leftarrow$ `1` when the processor sees a specific instruction repeated two times in a row. Not shown here are the two processes used to track when this instruction has been seen. The attack circuit we inserted is shown in Figure 5. We inserted the line of code shown there as a replacement to the code shown in Figure 4. As in our simple example (Figure 2), in this circuit there is no pair of dependent signals that could have been replaced by a wire during design-time testing without changing the behavior of the output signal. In the absence of the "knock" instructions, the attack code works as the original code did. For additional confidence that our modified processor could pass design-time testing, we tested it using the official SPARC certification test benches from SPARC International. The processor passed all tests, allowing the malicious process to carry the SPARC trademark.

In total, we inserted 19 lines of code into the processor. This includes the code to set the supervisor bit as well as the code to track the presence of the trigger instructions. We also removed 7 lines of the original code from the processor, for a net increase of 12 lines of code. This is comparable to the number of changes required for the attack used in the UCI paper.

Using the same tools and environment as in the UCI paper, we performed UCI analysis on the processor pipeline augmented with our attack. We used the test bench that comes bundled with the Leon3 processor for the UCI analysis. While the original UCI experiments used additional test cases, adding additional test cases does *not* increase the chance of detection by UCI. The number of circuits flagged by UCI as potentially malicious only *decreases* with the use of additional test cases. We discuss this point further in Section VI.

UCI was unable to detect our attack, with one caveat, which the next section describes.

C. Analysis of UCI Results

The caveat is that, to test our attack, we extended the test bench slightly. The instruction we used as our trigger did not appear anywhere in the original test bench. Therefore, UCI was able to detect that the `instruct_prev` and `instruct_curr` signals (which indicate whether the previous/current instruction is the trigger instruction) were constant at 0. We added one instance of this instruction to the test bench, and confirmed that UCI is then unable to detect any sign of the attack circuit.

This modification to the test bench violates our assumption that the attacker can not control the tests used during design-time testing. However, as we argue next, this is not an inherent limitation of our attack, but rather an accidental artifact of a sub-optimal implementation of the attack, and it would be straightforward for an attacker to adjust the attack to avoid this issue. In particular, it would be easy to pick some other instruction that appears at least once in the test bench, but that does not appear twice in a row. It seems reasonable to assume that the attacker would know the

[4]`holdn` is an active low signal that depends on a similar hold signal in the data and instruction caches

contents of the test bench, or at least some of the test cases, making it easy to choose a trigger instruction that already exists in the test bench. In retrospect, we chose our triggering instruction poorly: we overlooked the need to choose a trigger instruction that already occurs in the test bench. It would have been trivial to choose a different instruction, and if we had chosen more wisely, no modification to the test bench would have been necessary.

We also verified that none of the false positives produced by UCI would inadvertently reveal our backdoor. The UCI algorithm produces a fair number of false positives: pairs of dependent signals that happen to always be equal during the design-time testing that was done, but that can be non-equal under legitimate, untested input configurations. If the attack circuit happens to fall within the dependency chain of a pair of such signals, the attack would be highlighted by UCI as potentially malicious. In other words, UCI might get lucky and catch the attack by sheer chance. If the system designers reviewed everything highlighted by UCI, then they might notice the attack circuit as they reviewed the false positives. To see if this was the case with our attack, we looked at all the pairs of signals highlighted by UCI and verified that none of the gates involved in our attack were involved in the dependency chain between two such signals.

We also verified that our attack is successfully able to defeat BlueChip, a hybrid software/hardware solution that builds upon UCI to defend against backdoors [2]. BlueChip removes any hardware flagged by UCI as potentially malicious and replaces it with a trap. During execution, any instruction that would have used the removed circuitry gets simulated in software using alternate instructions. Because our attack circuit is not detected by UCI, it is not removed by BlueChip, and thus BlueChip is not able to detect or defend against our backdoor. However, our attack relies upon the assumption that the trigger instructions will be executed natively, in hardware, and will not trap; we verified that this is indeed the case and that our trigger instructions will not cause an unwanted trap. In particular, we compared the set of (false positive) signals flagged by UCI when run on the processor using the regular test bench to that returned by UCI when using a test bench augmented with a test that executes our trigger instructions. There was no difference between the two. This demonstrates that our trigger instructions do not cause BlueChip to trap and simulate them in software; the instructions execute normally and successfully engage the backdoor.

To test our attack, we used an environment identical to that used in the UCI paper. We synthesized the Leon3 processor with our embedded attack on a Xilinx Virtex 5 FPGA development board and successfully booted the SnapGear Embedded Linux distribution (linux-2.6.21.1 kernel). Then, we wrote a user-level program that triggers the backdoor. The program was able to successfully transition the processor into supervisor mode, using the backdoor we built.

D. Multiple Triggers

The above example is not perfectly stealthy, since there is always a chance the two-input trigger condition could occur as part of the traditional design-time testing all circuits are put through. Arguably, if the trigger instruction is chosen carefully, the chance that the test suite includes some sequence where the trigger instruction is executed twice in a row (thus detecting the backdoor) is low, but there still remains some chance that the backdoor is accidentally exposed during design-time testing.

We note that a smart attacker can virtually eliminate the chances of accidental exposure by choosing a trigger condition that holds only for a vanishingly small proportion of inputs. If the attacker uses an n-bit trigger condition, there is only a $1/2^n$ probability that a randomly chosen test will expose the backdoor; by choosing n sufficiently large, the attacker can make the probability that any test reveals the backdoor negligible.

All of the malicious circuits discussed until now have only a 2-bit trigger. It is natural to wonder whether these attacks can be generalized to work with an n-bit trigger condition, for arbitrary n. Through manual analysis, we identified several malicious circuits that could be extended to an n-bit trigger condition, and we show one such circuit in Figure 6. This circuit has two non-trigger inputs $i = (i_0, i_1)$ and n trigger inputs $t = (t_0, t_1, \ldots, t_{n-1})$, with the trigger condition $t_0 \wedge t_1 \wedge \cdots \wedge t_{n-1}$. The output function of this circuit is $f = (i_0 \wedge i_1) \vee (t_0 \wedge t_1 \wedge \cdots \wedge t_{n-1})$. Under non-trigger conditions the output function is $f_{NT} = i_0 \wedge i_1$. Under trigger conditions the output is stuck at $f_T = 1$.

The attack from Section IV-B can be extended to use an n-bit trigger, thereby making the backdoor even harder to detect through design-time testing. Figure 7 demonstrates one way to do so. This figure shows a design where executing a single specially chosen instruction n times in a row triggers the backdoor. To make the figure cleaner, we don't show the sub-circuit $(\neg holdn \wedge super) \vee (holdn \wedge in.super)$ and instead show only its output. The circuit remains stealthy when the sub-circuit is put back in place. Other generalizations are possible; for instance, we could choose a secret sequence of n different instructions and trigger the backdoor only when those n instructions are executed consecutively. In this way, the attacker can make it vanishingly unlikely that the backdoor will be detected through black-box testing.

V. Possible Defenses

Given the two attack circuits covered so far, there are a number of modifications to the UCI algorithm it might be natural to suggest. In this section, we look at a number of possible fixes to UCI and discuss the validity of each one. We argue that none of the natural modifications to UCI is secure: they can all be defeated through simple variations on our attacks. This suggests that it will not be straightforward to develop effective defenses against these attacks.

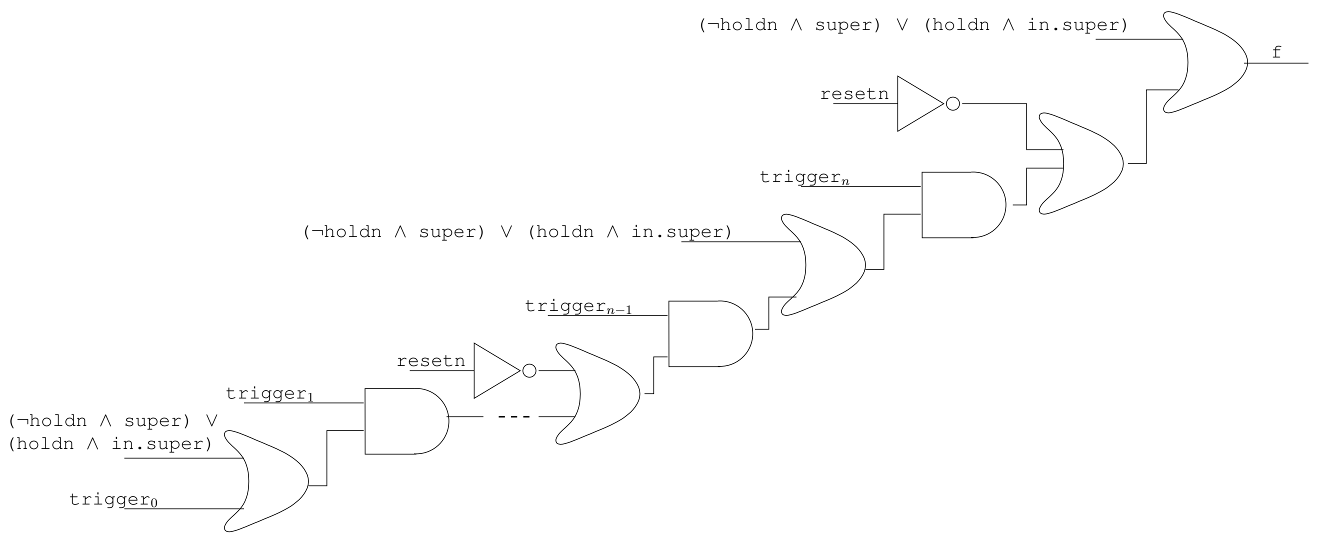


Figure 7. Practical attack circuit augmented to use n triggers.

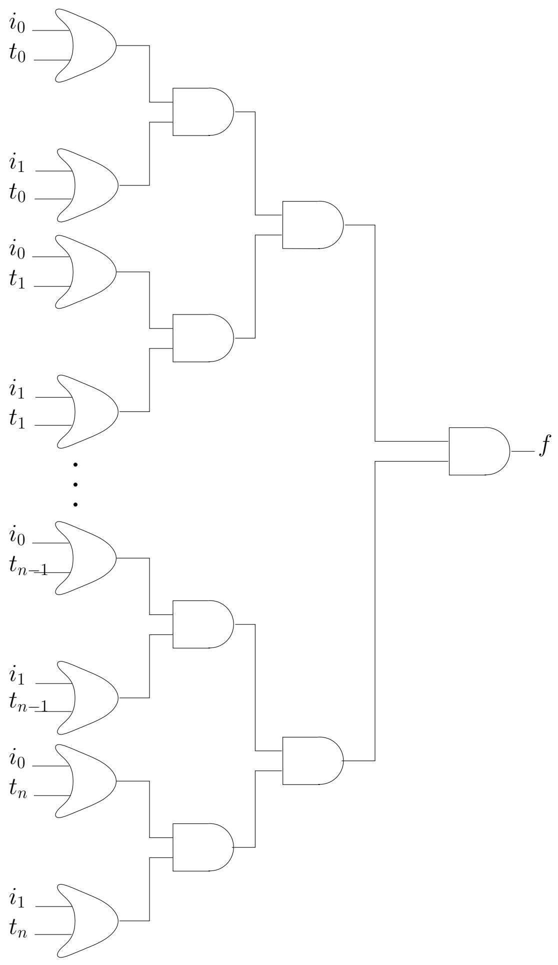


Figure 6. Stealthy and malicious circuit with n trigger inputs.

A. Handling of MUX Control Inputs

The UCI algorithm, as described in the original UCI paper [2, §5.3], does not treat the output of a MUX gate as data-dependent upon the control input to the MUX, and thus does not include these among the pairs of signals it analyzes, potentially missing some attacks. The original paper suggested that it would be straightforward to extend the UCI algorithm to treat MUX outputs as data-dependent upon their control signals (by converting MUX gates to their AND and OR gate representation and treating them as pure data flow elements).

This extension to UCI does not stop our attacks. The attack circuit in Figure 5 does not use any MUX gates, and thus is not affected by this modification to UCI: the attack remains effective and undetected by UCI. Therefore, our backdoor for the Leon processor would be unaffected by this extension to the UCI algorithm.

The example in Figure 2 does use MUX gates, but as it happens, even the extended UCI algorithm does not flag it as malicious, since there are test cases where the MUX outputs differ from their control inputs (e.g., $(0, 0, 0, 1)$, $(1, 0, 1, 0)$). Our search algorithm considers the output of every gate to be dependent upon all of its input signals, without differentiating between control vs. data flow, so every stealthy circuit output by our search algorithm already defeats the extended UCI.

B. Shrinking the Basis of Gates

The example shown in Figure 2 uses two MUX gates. One might wonder if the reason UCI fails to detect the malicious circuit is because the MUX gate is a macro element that contains three basic operations. It would take two AND gates and one OR gate to compute h, the intermediate signal computed by the first MUX gate. It feels as though the MUX gate is hiding some intermediate signals that might be present once this gate is compiled to silicon. It is natural to conjecture that, if these intermediate signals were exposed to UCI, UCI might be able to detect the attack. For instance, if we restrict the circuit so that it can only use simple gates such as $\{\text{AND}, \text{OR}, \text{NOT}\}$, then perhaps it would not be possible to defeat UCI (one might hope).

This defense does not work. As our practical attack on the Leon3 processor shows, it is possible to build a circuit that defeats UCI using only that three-gate basis. Moreover, with an additional search, we were able to find admissible, malicious, and stealthy circuits that used only NAND gates, as well as attacks that used only MUX gates. This demonstrates that even if we compile the circuit to NAND gates and then run UCI on the result, UCI still does not detect all malicious circuits.

We also performed $\binom{5}{2}$ additional searches with an expanded basis, to examine each two-gate basis possible from the set $\{\text{AND}, \text{OR}, \text{NOT}, \text{NAND}, \text{2-input MUX}\}$. We found admissible, malicious, and stealthy circuits for each of these $\binom{5}{2}$ bases. This indicates that exposing additional intermediate values or shrinking the set of allowed gates is not sufficient to make UCI secure.

C. Replacing Circuitry with a Single Gate

For each of the malicious circuits shown in Section IV, there exists some sub-circuitry that could have been replaced with a single gate without changing the output under the non-trigger condition. The UCI algorithm looks for a dependent pair of signals where the intervening sub-circuit could be replaced with a wire. It seems natural to extend UCI to also look for any triplet of dependent signals (s, t, u) where u is dependent on both s and t and check whether the intervening circuitry between them could have been replaced by a single gate from the basis.

Unfortunately, this too can be broken. Figure 8 is a counter-example that shows this approach is insufficient. The circuit was built using the basis $G = \{\text{MUX}\}$. The output function of the circuit is $f = (i_0 \wedge \neg i_1) \vee (t_0 \wedge t_1 \wedge i_1)$. Under non-trigger conditions the output is $f_{NT} = i_0 \wedge \neg i_1$. Under trigger conditions the output is $f_T = i_0 \vee i_1$. There is no portion of the circuit that could be replaced with a single MUX gate without changing the function output under non-trigger conditions.

D. Expressiveness of Attacks

While the attack we demonstrated is very powerful, the function of the malicious circuit we used was very simple. (For instance, the attack in Figure 2 acts like a simple AND gate, in the non-trigger condition.) It is reasonable to ask how much the attacker is constrained by evading UCI. While we don't have any definitive answer to that question, the results from our search for admissible, malicious, and stealthy circuits suggests that there are enough distinct functions for the attacker to have a reasonable chance of implementing her malicious hardware of choice while still evading UCI.

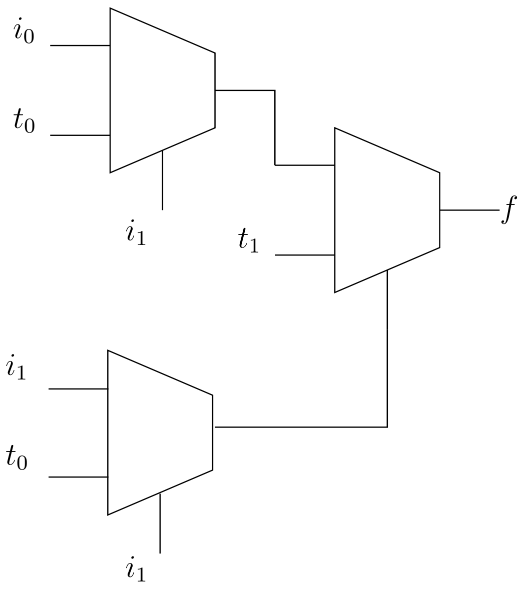


Figure 8. Stealthy and malicious circuit in which no portion of the circuit could be replaced with a single MUX gate.

For each search we explored all circuits up to a size of three gates. Once we had seen all circuits with less than four gates we stopped creating new circuits. Although our focus was on enumerating all circuits with at most three gates, as a consequence of our searching algorithm we ended up building some circuits with more than three gates along the way. Therefore our results do include some larger circuits.

We found 2,317 unique circuits that are admissible, obviously malicious, and stealthy. Of those, 75 had unique output functions. To put this number in perspective, the total number of unique output functions (for a circuit comprising any number of gates, but given our constraints of two non-trigger and two trigger inputs) that would meet our definition of obviously malicious and admissible (although not necessarily stealthy) is $2^4 \times 2^4 = 256$. The total number of unique output functions for any four-input circuit of any size is $2^{16} = 65536$. Of the 75 unique output functions we found, 27 could be expressed by a circuit with 3 or fewer gates. The circuits for the other 48 output functions contained between 4 and 7 gates. The smallest admissible, obviously malicious, and stealthy circuit we found used 2 MUX gates. The smallest circuit that did not use a MUX gate required 3 gates.

In addition to bounding the search, we also forced the search to time out after ten minutes, regardless of whether all one-, two-, and three-gate circuits had been explored. Therefore, 27 represents a lower bound on the number of unique output functions available when using up to three gates.

VI. Discussion

Our experience working with UCI revealed a counter-intuitive property of the algorithm. For a fixed circuit with a hidden attack, the chances that UCI detects the attack *decreases* as we increase the number of test cases in the test suite: the more thorough the test suite, the easier it becomes to defeat UCI.

To see why this is so, suppose we have a circuit that contains some malicious logic between signals s and t. Suppose there exists a single input x to the circuit that makes s and t non-equal and that doesn't fall into the trigger condition. For a reasonably complex circuit there will be thousands of non-trigger input configurations, not all of which will be used during design-time testing. If the input x is not found in the test suite, then UCI will (correctly) flag the logic between s and t as malicious. But if another test is added that does evaluate the circuit on input x, then there will now exist a test where s and t are non-equal and therefore UCI will (incorrectly) remove the flag, treating the circuit as benign. It is counter-intuitive, but we found that the number of false negatives (missed attacks) increases as one adds more test cases (and, conversely, the number of false positives decreases as the number of tests increases).

Of course, we must not lose sight of the bigger picture. The goal is to catch the malicious hardware by whatever means possible. Increasing the number of test cases increases the chances the attack is uncovered during traditional design-time testing. Each additional test case is an opportunity to detect that the hardware failed to behave as expected, so increasing the number of tests increases the effectiveness of design-time testing even as it decreases the effectiveness of UCI. In constructing malicious hardware, the attacker must walk a line between evading UCI and remaining hidden during design-time testing.

However, the n-bit trigger version of our practical attack demonstrates how the attacker may be reasonably confident of avoiding detection through design-time testing or UCI, regardless of the amount of testing conducted. This points to a larger challenge facing any UCI-like technique. Any malicious hardware detection scheme that uses test cases as its sole specification of correct behavior is working with an incomplete specification. It is impractical (and, in the case of a microprocessor, effectively impossible) for a test suite to exhaustively cover all possible test cases, and consequently a test suite provides only an incomplete specification of the intended behavior of the circuit. In the absence of a complete specification of the desired behavior, it is not clear how to define malicious behavior. (Given a complete specification, malicious behavior may be defined as any behavior falling outside the specification.)

Given this constraint, one approach for defending against malicious circuitry is to define a particular class of malicious hardware and then work to defend against that class. UCI implicitly defines one such class of circuits as the set of malicious circuits that will stay inactive during design-time testing. We were able to break UCI by finding malicious circuits that fall outside that class of circuits, i.e., by finding malicious circuits that are active during design-time testing but not detected by it. In short, our work demonstrates the class of malicious hardware UCI defends against is not comprehensive enough.

In general, the problem of malicious code detection is equivalent to the problem of proving the circuit is correct. The best solutions we have for tackling this problem at design time are exhaustive simulation and formal verification. However, with the current state of the art, neither of these techniques can feasibly be complete. Therefore, a challenge for any future work in the area of malicious hardware detection at design time is to clearly identify a class of malware to defend against and justify why this class is sufficient to capture every attack we might care about.

It is important to note that we made no attempt, nor do we claim, to define the class of all possible malicious circuits. We artificially imposed a significant constraint on our search for malicious circuits: in all of the attack circuits described in this paper, the input wires can be separated into trigger inputs and non-trigger inputs. Moreover, the definitions given in Section III consider only circuits where such a complete separation exists. We focused on this special class of circuits solely because it is easy to reason about algorithmically and it made it easier to find attacks. However, there is no reason to assume malicious backdoors will necessarily exhibit this separation in practice. Therefore, any proposed fix to the UCI algorithm or any future design-time algorithm for detecting malicious circuits should, like UCI, not assume that such a separation exists between trigger and non-trigger inputs.

VII. Related Work

There has been considerable prior research on methods for protecting against malicious hardware. We do not attempt to summarize all of that work here. Rather, our discussion of related work focuses on research involving the design-time insertion or detection of malicious hardware. There are other possible points along the hardware life-cycle where malicious logic could be inserted or detected (for example, the fabrication and supply chain stages [5]–[13]), but in this paper we focus on the RTL-level design stage, as that is the target of the UCI algorithm. Other stages have very different properties and constraints.

We first present attack-oriented research, then cover research on defense mechanisms, and lastly, highlight formal methods that may be applicable to the design-time attack model.

A. Hardware Attacks

Hadzic et al. were the first to look at what hardware attacks might look like and what they could do [14]. They specifically targeted FPGAs, showing it is possible to add malicious logic to the FPGA's configuration file that would short-circuit wires, driving them with logic high values. A wire with multiple high drivers increases the device's current draw, possibly causing the destruction of the device through overheating or wear-out failures. Hadzic et al. also proposed both a change to the FPGA architecture and a configuration analysis tool that would defend against the proposed attacks.

Agrawal et al. describe three attacks on RSA hardware as a part of a larger paper describing a defense against supply-chain attacks [15]. One of their attacks uses a built-in counter that shuts down the RSA hardware after a prescribed number of clock cycles. The other two attacks use a comparison based trigger that contaminates the results of the RSA hardware when activated. These attacks show that targeted hardware attacks can have a small footprint in terms of circuit area, power, and coding effort required.

The Illinois Malicious Processor (IMP) by King et al. [1] is the first work to propose the idea of malicious hardware being used as a support mechanism for attack software. These hardware security vulnerabilities, inserted during design time, are termed footholds. Since footholds can be introduced without changing many lines of code and without much effect on the rest of the design, they can be difficult to detect using conventional means or side-channel analysis. IMP demonstrates two attacks in particular: unauthorized memory access, which allows user processes to access restricted memory addresses, and shadow mode, where the processor executes in a special, hidden mode. As a part of IMP, King et al. showed how malicious software services can leverage the inserted footholds to escalate privileges, circumvent the login process, or steal passwords. In follow-on work, Hicks et al. [2] reimplemented the attacks and verified that the attacked hardware passed SPARCv8 certification tests.

Jin et al. developed eight attacks on the staged military encryption system codenamed Alpha [16]. The attacks corrupted four different units and three of the data paths of the encryption system. The eight attacks ranged in area overhead from less than 1% to almost 7% while still managing to pass design-time testing. The results highlight the risk of small, buried, but powerful attacks.

B. Defenses to Hardware Attacks

Research on detecting and defending against malicious hardware can be categorized into purely design-time methods and methods that involve a run-time aspect.

1) Design Time: Huffmire et al. study how to integrate untrusted IP cores with trusted IP cores by enabling architects to restrict communication between IP cores. They propose using areas of dead logic (moats) around each IP core and a verifiable inter-module communication philosophy (drawbridges) [17]. This approach ensures that no trusted IP core is contaminated or spied upon by an untrusted IP core, not even over a side channel. Because UCI and moats and drawbriges attempt to solve different problems, our attacks on UCI do not affect moats and drawbridges.

2) Run Time: Waksman et al. propose TrustNet and DataWatch [18]. Instead of preventing the inclusion of malicious circuits during design time as UCI attempts to do, TrustNet and DataWatch attempt to suppress malicious behavior during run time. The goal of TrustNet and DataWatch is to prevent untrusted hardware units in a processor's core pipeline from leaking information (i.e., producing too much output information) or stopping the flow of information (i.e., producing too little output information). For each input value to an untrusted pipeline stage, the pipeline stages before and after the untrusted stage determine if the output produced by the untrusted stage contains exactly the expected amount of information (e.g., when multiplying two 32-bit operands, was the result precisely 64 bits wide?). Because only the amount of output data is checked, TrustNet and DataWatch fail to detect incorrect output values of the correct width. Consequently, TrustNet and DataWatch are vulnerable to malicious backdoors that tamper with the results of computations without affecting the size of those results, so both TrustNet and DataWatch and UCI can be defeated if the attacker chooses the backdoor appropriately.

C. Formal Analysis of Hardware

Hardware, due to its limited resources and cycle-based behavior is generally quite amenable to formal analysis. Currently, the majority of research on formal methods, as applied to hardware, focuses on verifying that the hardware faithfully implements a given specification.

Model checking is one formal verification technique that verifies the behavior of a hardware design satisfies a set of properties, which are specified using temporal logic formulas [19], [20]. The verification is done through a bounded exhaustive search of the design state space. However, model checking is limited in its ability to scale to complex designs due to the size of the state space to be explored, which is exponential in the size of the state [21]. One approach that partly addresses the state-space explosion problem is to apply model checking to an abstract model of the processor [22], but many challenges remain.

Another formal verification technique is to develop a proof that an abstract model of the processor behaves as prescribed by the given specification [23]–[28]. The proof may be developed by hand and verified by a checker, or developed interactively with a theorem prover.

Given sufficient time, computational power, and a complete specification, formal verification techniques can be a powerful tool for detecting malicious circuits. Even without unlimited resources or a complete specification, defenders might reasonably apply formal methods to the problem of detecting malicious backdoors in processors. It is possible to apply formal verification to only certain security-critical modules of the processor. If these modules are simple enough that a complete specification can be written for them, it may be possible to fully formally verify such modules, thereby assuring they are free of backdoors. For instance, it would have been possible to detect our backdoor from Section IV by formally verifying the logic for when to transition to supervisor mode without full formal verification of the entire processor. The use of formal verification is not guaranteed to detect all malicious backdoors, but it might make it harder for an attacker to successfully evade detection.

VIII. Conclusion

We demonstrate an attack against UCI, a recently proposed algorithm for malicious hardware detection. UCI attempts to detect malicious hardware inserted at design time by identifying pairs of dependent signals in the source code that could seemingly be replaced by a wire without affecting the outcome of any test cases. Experiments show that it is possible to build malicious circuits in which no two dependent signals are always equal during design-time testing, yet where the circuit exhibits hidden behavior upon receiving a special input, called the trigger input. Using these circuits, we implement a malicious backdoor in the Leon3 processor that UCI is unable to detect. The attack allows a user-level program, with knowledge of the secret trigger, to enter supervisor mode, bypassing any OS-level checks. This work demonstrates that detecting malicious backdoors in hardware remains an open problem and suggests that it may not be easy to devise a reliable algorithm for detecting such attacks.

IX. Acknowledgment

We thank our shepherd Kevin Fu for his guidance and we thank the anonymous reviewers for their feedback and suggestions. This research was funded in part by National Science Foundation grants CCF 0811268, CNS 0953014, and CCF 0424422 and by AFOSR MURI grant FA9550-09-01-0539. Any opinions, findings, conclusions or recommendations expressed in this paper are solely those of the authors.

References

[1] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, “Designing and implementing malicious hardware,” in *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008, pp. 1–8.

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

[3] J. Markoff, “Old trick threatens the newest weapons,” *The New York Times*, p. D1, October 27 2009.

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

[5] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, “Silicon physical random functions,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 148–160.

[6] B. Moyer, “A PUF piece: Revealing secrets buried deep within your silicon,” *EE Journal*, January 24 2011, http://www.techfocusmedia.net/archives/articles/20110124-puf/.

[7] D. Du, S. Narasimhan, R. S. Chakraborty, and S. Bhunia, “Self-referencing: a scalable side-channel approach for hardware trojan detection,” in *12th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2010.

[8] Y. Jin and Y. Makris, “Hardware trojan detection using path delay fingerprint,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008.

[9] R. Rad, M. Tehranipoor, and J. Plusquellic, “Sensitivity analysis to hardware trojans using power supply transient signals,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008.

[10] T. Kean, D. McLaren, and C. Marsh, “Verifying the authenticity of chip designs with the designtag system,” in *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008.

[11] A. Das, G. Memik, J. Zambreno, and A. Choudhary, “Detecting/preventing information leakage on the memory bus due to malicious hardware,” in *Design, Automation & Test in Europe*, 2010.

[12] M. Potkonjak, “Synthesis of trustable ICs using untrusted CAD tools,” in *Design Automation Conference (DAC)*. ACM/IEEE, 2010.

[13] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber, “Modeling attacks on physical unclonable functions,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[14] I. Hadžić, S. Udani, and J. M. Smith, “FPGA Viruses,” in *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*. Springer, 1999.

[15] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, “Trojan detection using IC fingerprinting,” in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007, pp. 296–310.

[16] Y. Jin, N. Kupp, and Y. Makris, “Experiences in hardware trojan design and implementation,” *IEEE International Workshop on Hardware-Oriented Security and Trust*, 2009.

[17] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[18] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[19] C. Seger, "An introduction to formal hardware verification," University of British Columbia, Vancouver, BC, Canada, Canada, Tech. Rep., 1992.

[20] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, 1999.

[21] R. Pelanek, "Fighting state space explosion: Review and evaluation," *Formal Methods for Industrial Critical Systems*, vol. 5596, pp. 37–52, 2009.

[22] V. A. Patankar, A. Jain, and R. E. Bryant, "Formal verification of an ARM processor," in *Twelfth International Conference On VLSI Design*, 1999.

[23] M. Srivas and M. Bickford, "Formal verification of a pipelined microprocessor," *Software, IEEE*, vol. 7, no. 5, pp. 52–64, 1990.

[24] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1994, pp. 68–80.

[25] J. U. Skakkebaek, R. B. Jones, and D. L. Dill, "Formal verification of out-of-order execution using incremental flushing," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 98–109.

[26] M. N. Velev and R. E. Bryant, "Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*. ACM, 2000, pp. 112–117.

[27] R. Hosabettu, G. Gopalakrishnan, and M. Srivas, "Formal verification of a complex pipelined processor," *Formal Methods System Design*, vol. 23, no. 2, pp. 171–213, 2003.

[28] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of TriCore2 and other processors," in *DVCon*, February 2007.