Considering Software Protection for Embedded Systems

Dr. Yong C. Kim and Lt. Col. J. Todd McDonald, Ph.D
The Air Force Institute of Technology

Software in modern embedded systems is often realized by using prefabricated reconfigurable computing devices such as Field Programmable Gate Arrays (FPGAs). Such devices support the use of portable hardware description languages and, as a result, have vulnerabilities consistent with normal software applications. In this article, we consider the nature of adversarial reverse-engineering attacks in this environment and measures of protection.

In our modern world, the meaning of a word can change quite often. Even the term computer previously referred to a human operator who crunches numbers while today we relate this term clearly to a machine. With the emergence of new reconfigurable computing technologies such as FPGAs, the definitions of software and hardware have become less clear. As Vahid suggests [1], we should stop calling circuits hardware and start broadening what we consider software.

In the traditional sense, software referred to the bits (1s and 0s) representing language statements that could be executed on hardware processors. Today, embedded systems utilizing FPGAs realize circuits merely by downloading a sequence of bits that instantiate gates, controllers, arithmetic logic units, crypto circuits, and even processors. Thus, a circuit implemented on embedded systems utilizing an FPGA is essentially software.

Considering the proliferation of embedded systems with reprogrammable hardware components in both commercial and military sectors, we can readily show the impact of malicious activity geared to reverse engineer, tamper, or copy critical technologies residing in those systems. In this article, we delineate protective transformations for such embedded logic and present a brief survey of reverse engineering attacks in this realm.

Characterizing Circuit Protection

Both the DoD and the commercial sector have an interest in describing and measuring candidate protective measures, whether they derive from hardware anti-tamper realizations or software-based techniques. Adequately defining criteria for successful software protection in practice remains elusive mainly because full protection may not be possible, at least theoretically [2]. Collberg and Thomborson [3] describe three practical means of protecting software against copying, reverse-engineering, and malicious tampering; these include, respectively, water-marking, obfuscation, and tamper-proofing. In terms of analyzing protection mechanisms, they suggest measuring obfuscating transformations based on their obscurity (how much time is increased for understanding and reverse engineering), resilience (difficulty for reversing the transformation), stealth (the natural context of the transformation), and cost (overhead).

Though embedded systems may encompass a wide variety of custom processors and components, our discussion focuses on more fundamental logic programs represented as combinations of gate-level logic. In describing such circuits, we use two primary analysis paradigms: how they behave, and how they are constructed. We express the black-box behavior of a circuit by enumeration of all inputs, subsequent evaluation and propagation of signals on all intermediate gates, and the recording of the corresponding output. Figure 1 illustrates an input/output representation of a small combinational logic circuit with three inputs (X1, X2, X3), four intermediate gates (4, 5, 6, 7), and two distinguished intermediate gates (Y6, Y7) known as outputs.

We define a signal as a vertical reading of a column in the truth table (a fully enumerated input/output behavior, based on canonical ordering of inputs) and call the signature of a circuit the collection of its output signals. Given the full truth table of a circuit, we define its gray-box behavior as signals of all intermediate logic gates based on the enumeration of all possible inputs.

The white-box structure of a circuit may be represented by textual description languages (Bench, Verilog, VHDL, etc.), which are regular grammars that support expression of gates, electrical signals, components, and gate interconnections. Textual representations translate into graphical forms, which are referred to as...
the circuit topology. Figure 2 illustrates
the circuit seen in Figure 1 in correspond-
ing graphical representation and a Bench
textual description [4]. We define a com-
ponent within the circuit as a collection of
lower-level elements (such as gates) that
form a distinct sub-circuit.

The semantics (or black-box behavior)
of a circuit consists of only the input and
output signal pairs (the X and Y signals
seen in Figure 1). Intuitively, one way to
think of circuit protection is the act hiding
all intermediate transitions that transform
input to output. The collection of these
transitions, in essence, represents the
intellectual property of a circuit. Without
knowledge of the original intermediate
transitions, no human or automated
process may derive other information
about the original circuit such as topology,
signal definitions, or component defini-
tions. Many define the essence of circuit
reverse engineering as the ability to cor-
correctly identify topology or components of
the original circuit [4, 5].

To protect a circuit, replace the origi-
inal circuit with a semantically equivalent
version (one which does the same func-
tion) that hides the intellectual property of
the original in some definable or measur-
able way. For the circuit in Figures 1 and
2, a replacement circuit would have ident-
tical signals for inputs and outputs (X1, X2, X3, Y6, Y7), but would have some
other internal white-box construction
(represented by gates 4 and 5 in Figures 1
and 2).

This formulation restates the essence of
a virtual black box [2] because it
defines full protection as a replacement
circuit that does not leak any more infor-
mation relative to an original circuit (other
than its input/output characteristics). In
more practical settings [3], the goal of
using a replacement circuit becomes
obscuring the original circuit in some way
so that the cost of reverse engineering is
maximized while operation characteristics
of the circuit are not degraded beyond an
acceptable level. Next, we delineate the
permissible transformations on a circuit
when obfuscation is in view.

**Characterizing Circuit Transformations**

We define an obfuscating transformation
$O(\cdot)$ as an efficient, terminating program
that takes circuit $P$ as input and returns
another circuit $P'$: $O(P) = P'$. Of this asser-
tion, all theoreticians and practitioners
(that we are aware of) would agree.

Beyond that, the majority of theoretical
and practical models for obfuscation have

---

**Figure 2: White-Box Circuit Description**

<table>
<thead>
<tr>
<th>INPUT(1)</th>
<th>INPUT(2)</th>
<th>INPUT(3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT(1)</td>
<td>OUTPUT(2)</td>
<td>OUTPUT(3)</td>
</tr>
</tbody>
</table>

Program $P$

Program $P'$

Input $x$

Input $x'$

Transformation

$\text{Transformation } s(P,k,x,y) = q,P',x',y'$

Output $y$

Output $y'$

Resolution

$y = q(y', k)$

---

**Figure 3: Black-Box Refinement**

---

We believe security may be provable in
some circumstances if we are allowed to
expand the semantic equivalence require-
ment (in other words, if an obfuscator can
change the (white-box) structure of a cir-
cuit so that (black-box) input/output rela-
tionships of the original circuit $P$ are also
changed). We refer to black-box transform-
ations with this meaning in mind. Likewise,
the obfuscator may change (white-box) structure in such a way so that
semantic equivalence with $P$ is preserved.

We refer to white-box transformation with this meaning in view.

**Black-Box Transformations**

Sander and Tschudin [6] were one of the
first to recognize the value of a black-box
transformation as a means to hide func-
tional intent. In discussing black-box
changes to $P$, we assume there are certain
programmatic environments where the
output of the obfuscated circuit $P'$ is con-
ductive for off-line analysis and, therefore,
open to the possibility of recovering the
intended output of the original circuit $P$. In
certain environments, this may not be
possible. Black-box transformations,
however, may be necessary to achieve
stronger guarantees of security. In order
to achieve a useful black-box transforma-
tion by some specific white-box changes
to the structure of a circuit, an obfuscat-
ing operation must meet two require-
ments:

1. **Change in Black-Box Behavior.**

   The functional behavior changes for
   some majority of values in the domain
   $x$, $P(x) \neq P'(x)$. This leaves open the
   possibility that some transformations
   may produce equivalent values for cer-
   tain values of $x$.

2. **Recovery of Black-Box Intent.**

   In order to recover the original functional
   output of $P$, some function $S(\cdot)$ must
   allow inversion: $V(x):P(x) = S(P'(x))$.

   One way of hiding or masking
   input/output relationships is to do so
   through transformation that keeps the
   input/output values hidden in plain sight.

   We refer to such techniques as a black-box
   refinement of the original circuit $P$ and
   present its algorithmic description in
   Figure 3. From the viewpoint of a circuit
   and its corresponding truth table, we can
   visualize at least five distinct operations
   that may be a part of a black-box refine-
We envision that all five would be applied in a probabilistic manner based on configurable properties found in a (secret) key. If we let \( X \) represent the domain of the original \( P \) and confine it to a fixed number of bits, a black-box refinement may do any of the following:

1. Add input bits so that a new domain with a larger possible bit string \( X' \) is created.
2. Randomly permute the input bits themselves, resulting in a virtual reordering of the bits.
3. Introduce intermediate gates that would result in new truth table columns for \( P' \).
4. Introduce a random number of output gates.
5. Randomly permute any of the output bits themselves.

Changing the full input/output relationships of a circuit may truly hide the original black-box intent of a circuit. By composing a circuit with a semantically strong data encryption algorithm, the resulting program exhibits input/output relationships with desirable cryptographic properties. Figure 4 depicts this black-box change, known as a semantic transformation.

**White-Box Transformations**

We define a structural white-box change to a circuit as a change to the topology of the underlying directed acyclic graph, which represents the circuit. Topological changes may involve textual renaming of signals or gates, changing the Boolean function type of particular gates, reordering input or output signals, introducing additional inputs, introducing additional outputs, concatenating the serial composition of the entire circuit with another circuit, merging the parallel composition of the circuit with another circuit, or replacing one or more gates within the circuit with a functionally equivalent set of gates.

Figure 5 shows the traditional meaning of obfuscation as understood in both theoretical and practical study: A transformation \( w(P, k) = P' \) takes as input a circuit \( P \) with some (possibly) probabilistic information embodied in key \( k \). We consider any random choices made by an obfuscation process to be part of this key. The output of \( w(\cdot) \) is a circuit \( P' \) that remains functionally equivalent to the original circuit \( P \) and represents a different version of the original. Current obfuscation research centers on the transformation algorithm and defining the security that is achieved by its use.

**Reverse-Engineering Attacks**

In the world of real circuit analysis, the typical goal of a reverse engineer is to recover the input/output of the circuit in question by some method less than full exponential enumeration. As we have already alluded to with black-box refinement or semantic transformation, such transformations would (at a minimum) prevent this form of reverse engineering while simultaneously introducing the need for output recovery in order to maintain functional utility. There are a number of different ways to discover and alter the functionality of a circuit. The term tampering refers to broad categories of circuit exploitation, including subversion, modification, and reverse engineering. Reverse engineers typically target reproduction of a circuit’s functionality, usually for capital gain or malicious intent. Specific attacks can be roughly categorized as brute force, white-box/gray-box, side-channel, and fault injection.

**Brute Force Attacks**

Brute force attacks are based on black-box circuit behavior and are performed either while the circuit is in its natural environment or standalone in a simulator. Such attacks can be categorized as either general or passive.

- **General black-box attacks.** Traditionally, black-box attacks are the first and simplest means to reverse engineer a circuit. Adversaries glean black-box behavior by enumerating all possible input combinations and recording corresponding outputs. Using a large truth table, data analysis algorithms—or in some cases visual inspection—the adversary may re-create the underlying Boolean equations that define the circuit’s logic; this type of attack works well on circuits with well-defined inputs and outputs.

There exists potentially \( 2^n \) input combinations to fully characterize any combinational circuit and potentially \( 2^n + m \) or more input combinations for sequential circuits with \( m \) sequen-
techniques are the only way to get direct examination by looking at transistor states. Adversaries essentially use pictures to observe signals that are propagated by means of applied input values.

**Side-Channel Attacks**
We observe that even circuits that may be provably secure according to a theoretical model—based on static white-box and dynamic black-box behavior—may still leak critical information relative to the circuit’s function (based on real-world implementation issues). Rather than use brute force (to glean black-box behavior) or physically probe the internals of a circuit (to glean white-box and gray-box behavior), side-channel attacks use secondary information to create a picture of circuit functionality. Side channels are areas of a circuit that leak unintended information. They include power consumption and timing analysis:

- **Power Consumption.** Power consumption attacks mainly focus on breaking cryptographic schemes. The concept is that through an examination of the power used by a circuit, the underlying encryption algorithm can be found. This approach gives an attacker insight into the data values that are being manipulated on a chip. It is possible to then correlate this collected data to known functions in order to see exactly what is happening.

- **Timing Analysis.** With brute force attacks, synchronous circuits add additional complexity in the reverse-engineering process due to the timing constraints that are introduced. Timing attacks focus on taking the circuit outside of normal parameters by modifying the speed of the clock, either speeding it up or slowing it down. Because timing is linked directly to real-world physical implementations of various circuit technologies, our existing obfuscation framework requires additional information regarding structural characteristics of the circuit implementation.

**Fault Injection**
Fault injection is a generic term describing the injection of faults into digital systems using a variety of attacks: raising voltage higher or lower than system tolerances, including voltage spikes, or introducing clock glitches. An adversary may use any of these methods to cause the system to malfunction with intentions of revealing information useful in further attacks. The adversary performs fault injection dynamically at circuit run-time combined with power analysis techniques. Encryption algorithms, such as the Advanced Encryption Standard (AES), provide strength against brute-force key discovery from black-box behavioral analysis. However, an adversary may use fault injections with realized AES circuits in order to reduce encryption strength via key-space reduction. This exploit requires internal circuit access and reduces the goal of the adversary from using brute-force methods to interrupt the successful encryption/decryption process itself.

**Conclusion**
Given the current trend of reprogrammable embedded devices within the DoD and industry, attention needs to be focused on the benefits or measurability of software protection applied to this domain. Modern reconfigurable embedded systems now require us to consider circuits as software and the tamper methods applicable to physical circuits as new threats to a broadened definition of software. This article has presented a brief overview of the characteristics, transformations, and attacks possible in the realm of software implemented as circuits on an embedded system. Ultimately, we must turn our attention to the protection of critical technology resident in such an embedded system, mindful of the possible threats and techniques at our disposal.

**References**


**Note**

1. The views expressed in this article are those of the authors and do not reflect the official policy or position of the U.S. Air Force, DoD, or U.S. government.

---

**About the Authors**

**Yong C. Kim, Ph.D.**, is an assistant professor in the department of electrical and computer engineering at the Air Force Institute of Technology (AFIT). He received his doctorate in electrical engineering from the University of Wisconsin-Madison (UW-M), his master's degree in computer engineering from the UW-M, and his bachelor's degree in computer engineering from the University of Washington.

**Lt. Col. J. Todd McDonald, Ph.D.**, is a Lieutenant Colonel in the USAF and an assistant professor in the department of electrical and computer engineering at the AFIT. He received his doctorate in computer science from Florida State University, his master's degree in computer engineering from the AFIT, and his bachelor's degree in computer science from the USAF Academy.

---

**Be a CROSSTALK Backer**

CROSSTALK would like to thank the accompanying organizations, designated as CROSSTALK Backers, that help make this issue possible.

CROSSTALK Backers are government organizations that provide support to forward the mission of CROSSTALK. Co-Sponsors and Backers are our lifeblood.

**Backer benefits include:**

- An invaluable opportunity to share information from your organization's perspective with the software defense industry.
- Dedicated space in each issue.
- Advertisements ranging from a full to a quarter page.
- Web recognition and a link to your organization’s page via CROSSTALK’s Web site.

Please contact Kasey Thompson at (801) 586-1037 to find out more about becoming a CROSSTALK Backer.