Chapter 3
Logic Modification-Based IP Protection Methods: An Overview and a Proposal

Brice Colombier, Lilian Bossuet and David Hély

3.1 Introduction and Context

Design data protection schemes can be classified into two categories. *Passive* ones detect that counterfeiting of over-usage took place, but do not stop it. Conversely, *active* protection schemes actually prevent the infringement to occur in the first place. They do so by modifying the design in order to make it resilient to such threats. Chapter 2 gives an overview of the available features in most electronics design that can be turned into powerful locks. The current chapter focuses on protection means that require a modification of the combinational logic.

The first feature which can be achieved by logic modification is to controllably disturb the outputs. This allows the designer to make the circuit unusable. In order to return to normal operation, a "key" must be provided to the circuit. We use the word "key" here as a generic term, not as a cryptographically strong sequence of digits. This key is provided by the designer, who can therefore record how many times the key has been delivered, and how many instances of the circuit are activated. Such "count of the produced ICs" is called hardware metering [1]. It can be achieved by different means, which are presented in the next section.

The second feature which can justify logic modification is to slow down reverse-engineering. By adding extra logic gates, recovering the circuit functionality from a high-resolution picture [2] of the layout or a netlist can become extremely difficult. Logic obfuscation is one of the ways to do this, and is also presented.

B. Colombier (✉) · L. Bossuet
Hubert Curien Laboratory, UMR CNRS 5516, University of Lyon, Saint-Étienne, France
e-mail: b.colombier@univ-st-etienne.fr

L. Bossuet
e-mail: lilian.bossuet@univ-st-etienne.fr

D. Hély
LCIS, Grenoble Institute of Technology, Valence, France
e-mail: david.hely@lcis.grenoble-inp.fr

L. Bossuet and L. Torres (eds.), *Foundations of Hardware IP Protection*,
DOI 10.1007/978-3-319-50380-6_3

One of the key features of all logic modification-based protection schemes is the selection of the sites to modify. Those sites are the ones on which extra gates will be inserted. Different techniques can be used to this end, such as random selection [3], fault-analysis [4], etc. A trade-off between efficiency and computation time must be done by the designer. For example, finding the best place to insert a masking gate can be very time consuming, as shown in [4]. A novel technique which uses graph-analysis methods is presented. It selects the sites to modify orders of magnitude faster than fault analysis-based techniques, yet achieving better outputs disturbance than simple random selection.

Finally, all the schemes mentioned above need to be integrated in a complete design protection module. Indeed, even though many previous works try to exhibit security features in their protection schemes, such security can only be reached by using a dedicated cryptographic function. This is discussed in more details in the final section.

This chapter is organized as follows. In Sect. 3.2, we provide a formal framework for logic modification-based protection schemes by defining logic encryption, logic obfuscation, logic masking, and logic locking and give examples for each. In Sect. 3.3, we present a new graph-based algorithm that selects the optimal nodes to be modified to achieve logic locking of a combinational netlist. In Sect. 3.4, we present the results of implementation, specifically the logic resources overhead and analysis time. In Sect. 3.5, we evaluate the proposed method and develop associated metrics. In Sect. 3.6 we describe a threat model and perform a security analysis of the protection schemes considered. In Sect. 3.7, we discuss design considerations. In particular, we emphasize the need to introduce a cryptographic primitive to ensure security, and to not rely on the logic/masking module to fulfill this objective.

3.2 A Formal Foundation for Logic Protection Schemes

An increasing number of works are trying to find a way to protect the intellectual property of IP designers and fabless IC designers by acting on combinational logic. Unfortunately, most of these works make incorrect use of the terminology, i.e., *logic encryption*, *logic obfuscation*, *logic masking* and *logic locking* are used without a formal definition. This chapter takes the opportunity to propose a formal foundation for logic protection schemes. In this section, we provide formal descriptions and definitions of the logic protection schemes in order to strictly evaluate their different contributions to the literature. In all the following subsections, the original (not protected) n-input, l-output logic function is formalized by a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$.

3.2.1 Logic Encryption

The term "logic encryption" is used when a specific symmetric encryption function ξ_f over $GF(2^l)$ is applied to f. Formally, it is not *logic encryption*. The term is not specific. *Encryption of the Boolean function* f is the correct expression. The result of this encryption is the Boolean function $f'\{0,1\}^n \rightarrow \{0,1\}^l$. f' is given by the following expression, where k is the secret key:

$$f' = \xi_f(f, k)$$

ξ_f is a symmetric encryption function if and only if an inverse function ψ_f exists that uses the same secret key k for decryption, and is defined as follows:

$$\psi_f(f') = \psi_f(\xi_f(f, k), k) = f \tag{3.1}$$

Functions ξ_f and ψ_f must meet the following requirements:

$$\forall(k_i, k_j) \in (\{0,1\}^m, \{0,1\}^m), k_i \neq k_j$$

$$\xi_f(f, k_i) \neq \xi_f(f, k_j) \tag{3.2}$$

$$\psi_f(\xi_f(f, k_i), k_i) \neq \psi_f(\xi_f(f, k_i), k_j) \tag{3.3}$$

Functions ξ_f and ψ_f also have to satisfy the following requirements, where *Corr* is the function that computes Pearson's correlation coefficient.

$$\forall k \in \{0,1\}^m : Corr(\xi_f(f, k), f) \simeq 0 \tag{3.4}$$

$$\forall k \in \{0,1\}^m : Corr(\psi_f(\xi_f(f, k), k), \xi_f(f, k)) \simeq 0 \tag{3.5}$$

One of the consequences of the last expression is that the mean of the Hamming distance between the input and the output of the encryption/decryption functions is close to 50 % (ideally exactly 50 %) as described by the following expressions when the mean of the Hamming distance is computed for all the inputs of the Boolean function f:

$$\forall k \in \{0,1\}^m : \frac{\sum HD(\xi_f(f\{0,1\}^n, k), f\{0,1\}^n)}{2^n - 1} \simeq 50\,\% \tag{3.6}$$

$$\forall k \in \{0,1\}^m : \frac{\sum HD(\psi_f(\xi_f(f\{0,1\}^n, k)), \xi_f(f\{0,1\}^n))}{2^n - 1} \simeq 50\,\% \tag{3.7}$$

Some works [4–6] consider this last property as proof of security. This is a mistake, since it is possible to obtain the same result with a function that does not achieve

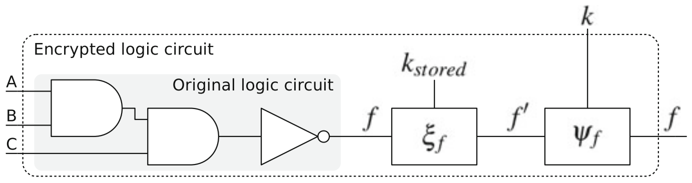


Fig. 3.1 Example of logic encryption

encryption. For instance, inverting the first $n/2$ bits of the output of f leads to a 50 % Hamming distance. Similarly, inverting every input of odd order leads to the same result. In both cases, the mean of the Hamming distance as described in (3.6) is equal to 50 % but the correlation defined in (3.4) is not zero.

These works are presented as "logic encryption," even though this is absolutely not the case. The authors of these works defined "logic encryption" as: "*logic encryption hides the functionality and the implementation of a design by inserting some additional gates called key-gates into the original design*" [5]. With this definition, logic encryption does not respect the expressions (3.1) to (3.7). Consequently, we claim that all works presented as "logic encryption" are inaccurate because in fact, they only propose to *mask* the logic functionality. The security level of such masking functions is very low compared with proper encryption.

A didactic example of true "logic encryption" is given by considering the following 3-input Boolean function $f\{0,1\}^3 \rightarrow \{0,1\}^1$:

$$f(A,B,C) = \overline{A.B.C}$$

Figure 3.1 is a diagram of the encrypted logic circuit. This includes the original logic circuit which computes the Boolean function f, the encryption function ξ_f which computes the encrypted Boolean function f' using an embedded secret key k and the decryption function ψ_f which outputs the correct result of the Boolean function f if and only if the correct key k is applied on the external key input.

This didactic example shows that the area overhead of true logic encryption is always prohibitive since it requires the implementation of encryption and decryption functions. Note that the security level of such a protection depends on the key size. Nowadays a secure implementation of a symmetric cipher has to use at least a 128-bit key. All protection schemes that include a secret key that has only a few bits fail to provide the designer with any security because of the feasibility of a brute force attack.

3.2.2 *Logic Obfuscation*

Logic obfuscation comes from the field of computer science in which developers wish to protect source codes against unauthorized reading and understanding.

The following definition of code obfuscation is proposed by Hachez [7]: *Transform a program P into another program P' harder to reverse engineer with the same observable behavior. If P fails to terminate or terminates with an error, then P' fails to terminate or terminates with an error. Otherwise, P' must terminate and produce the same output as P*. Hardware obfuscation consists in applying this definition to the hardware field, by changing the logic, FSM, or other part of a design without changing the system behavior.

When the logic part of a circuit is obfuscated, a design modification γ_f is applied to f. The result of this design modification is the Boolean function $f''\{0,1\}^n \rightarrow \{0,1\}^l$.

$$\gamma_f(f) = f''$$

The function γ_f must meet the following requirement for any input $x \in \{0,1\}^l$:

$$\forall x \in \{0,1\} : f''(x) = f(x) \tag{3.8}$$

Some works present logic obfuscation but do not fulfill requirement (3.8) [8, 9]. Most of these works use a secret key that changes the behavior of the original logic function. These works are typical cases of logic masking, which is presented in Sect. 3.2.3.

It is possible to try to perform obfuscation at the logic-gate level but this usually implies a large overhead. Indeed, obfuscation techniques aim to increase reverse-engineering time. The time is at least linear with the area [10]. Increasing the area increases the time needed for reverse engineering. As a consequence, the main design modification rule for obfuscation is to not follow the usual design rules for efficient implementation of a Boolean function. Usually, laws and theorems of Boolean logic are applied to Boolean functions in order to reduce the number of gates (i.e., the area) of the final hardware implementation. To obfuscate an implementation of a Boolean function, these laws and theorems are followed in the opposite way, i.e., they increase the size of the hardware implementation.

Two strategies are used in the first step of obfuscation: *develop* and *obscure*. To develop a Boolean function, the designer can use the canonical disjunctive normal form (also called *min-term* canonical form) in which the Boolean function is represented and implemented as a sum of *min-terms*. As a didactic example, let us consider the following 3-input Boolean function $f\{0,1\}^3 \rightarrow \{0,1\}^1$:

$$f(A,B,C) = \overline{A.B.C}$$

This Boolean function could be developed using the following canonical disjunctive normal form (first obfuscation step).

$$f''(A,B,C) = A.\overline{B}.\overline{C} + \overline{A}.\overline{B}.\overline{C} + \overline{A}.B.\overline{C} + A.B.C + A.\overline{B}.C + \overline{A}.\overline{B}.C + \overline{A}.B.C$$

f and f'' follow requirement (3.8). Figure 3.2a, b show the logic diagrams of the two functions with only 2-input AND and OR gates and inverters (other types of gates could also be used).

In order to obscure a Boolean function, the designer can apply to f'' some of the Boolean logic laws (absorption, complementary, common identities, etc.) and DeMorgan's theorem to increase the number of gates used in the hardware implementation. For example, by also using some redundant logic operations, f'' is described by the following Boolean expression:

$$f''(A,B,C) = A.\overline{B} + \overline{A}.\overline{B} + \overline{A}.B + \overline{B}.\overline{C} + \overline{A}.\overline{C} + A.C + \overline{B}.C + \overline{A}.C + B.C + \overline{A} + \overline{B} + C + \overline{A.B} + A \oplus C + A \oplus B + \overline{A.\overline{C}} + \overline{B.\overline{C}}$$

Again f and f'' follow requirement (3.8). Figure 3.2c shows the logic diagram of f'' after this second step of obfuscation. The designer can also insert dummy logic to further increase the reverse engineering effort.

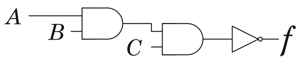
Table 3.1 shows the logic resources required for each logic circuit in Fig. 3.2. For each circuit, the number of gates is shown for each type (inverter, 2-input *and* gate, 2-input *or* gate and 2-input *xor* gate), along with the gate equivalent metric. The area overhead is given for the two hardware implementations of f''. As mentioned above, the increase in reverse-engineering time for each obfuscated logic circuit (in comparison with the original logic circuit) is supposed to be equal to the area overhead. For example, the time required to reverse engineer circuit shown in Fig. 3.2c is 14.58 times greater than the time required to reverse engineer the original circuit.

Due to the high area overhead, such logic obfuscation is not suitable for most applications. Moreover, the hardware design of the obfuscated circuit has to be performed by hand to avoid logic optimization by the synthesis tool. It is possible to mix a light logic obfuscation with obfuscation at another level. Indeed, hardware obfuscation is also possible at the HDL [11, 12] or layout levels [13, 14].

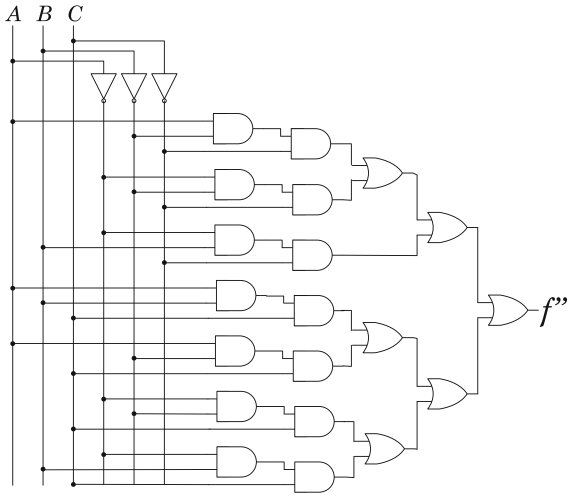
The above description of logic encryption and logic obfuscation allows us to affirm that none of the published works that present "logic encryption" or "logic obfuscation" meet the formal requirements of these two techniques. Most of these works in fact describe "logic masking" or "logic locking." In the remainder of this section we present logic masking and logic locking techniques.

3.2.3 *Logic Masking*

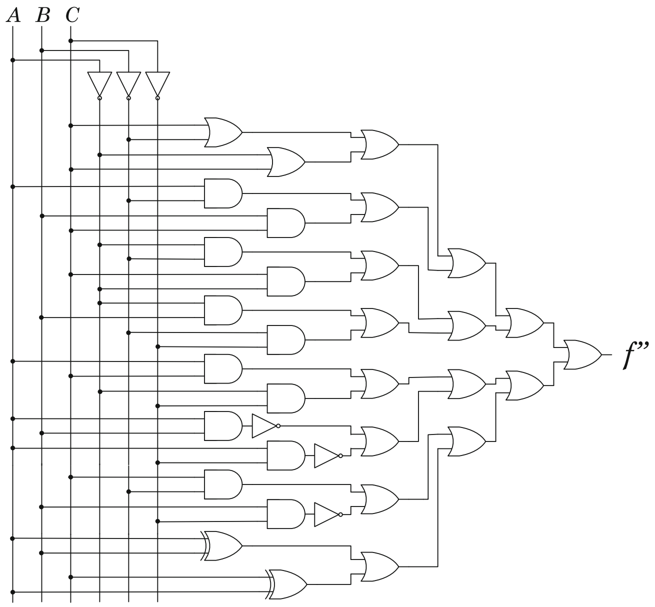
Logic masking consists in inserting *xor* or *xnor* gates in the data path of the logic circuit of a Boolean function in order to change the logic behavior of the circuit if the wrong masking key is applied. It was first proposed in [3]. Let us consider that a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$ could be represented as a set of i Boolean subfunctions $\{f_0, f_1, \ldots, f_{i-1}\}$. Logic masking of the Boolean function f by using the i-bit



(a) Original Boolean function implementation



(b) Boolean function implementation after a first step of logic obfuscation



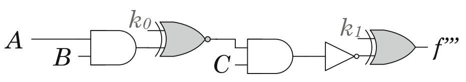
(c) Boolean function implementation after a second step of logic obfuscation

Fig. 3.2 **a** Original boolean function implementation, **b** Boolean function implementation after a first step of logic obfuscation, **c** Boolean function implementation after a second step of logic obfuscation

Table 3.1 Logic resources requirements and timing overhead for reverse-engineering of the circuits described in Fig. 3.2

Boolean function	Logic circuit	#Logic gates				Gate equivalent	Area/reverse-engineering time overhead
		inv	*and*	*or*	*xor*		
f	Figure 3.2a	1	2			4.01	–
f after first step of obfuscation	Figure 3.2b	3	14	6		35.41	+883 %
f after second step of obfuscation	Figure 3.2c	6	12	17	2	58.47	+1458 %

Fig. 3.3 Example of logic masking



masking key $k = \{k_0, k_1, \dots, k_{i-1}\}$ is described by the following expression, where f''' is a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$ and $\ominus$ is the *xor* or *xnor* Boolean operator:

$$f''' = \{f_0 \ominus_0 k_0, f_1 \ominus_1 k_1, \dots, f_{i-1} \ominus_{i-1} k_{i-1}\}$$

$$\forall j \in [0, i-1] \begin{cases} \text{if } \ominus_j \equiv xor \rightarrow k_j = 1 \rightarrow f_j \ominus k_j = f_j \\ \text{if } \ominus_j \equiv xnor \rightarrow k_j = 0 \rightarrow f_j \ominus k_j = f_j \end{cases} \tag{3.9}$$

The correct masking key k is found by using the laws in (3.9), and considering the type of inserted gate. As a didactic example, let us consider the following 3-input Boolean function $f\{0,1\}^3 \rightarrow \{0,1\}^1$:

$$f(A,B,C) = \overline{A.B.C}$$

This Boolean function could also be described by the following expression:

$$\begin{cases} f(A,B,C) = f_1(f_0(A,B),C) \\ f_0(X,Y) = X.Y \\ f_1(X,Y) = \overline{X.Y} \end{cases}$$

A didactic example of logic masking of the Boolean function f is given in Fig. 3.3, where $\ominus_0$ is an *xnor* gate and $\ominus_1$ is an *xor* gate. According to the laws in (3.9), we can determine the correct masking key $k = \{0, 1\}$ needed to obtain the original logic behaviour. In Fig. 3.3, additional masking gates are in gray.

Efficient insertion of the masking scheme has to be achieved without reducing performance (mainly by limiting the insertion of gates on the critical path) or increasing area overhead (by limiting the number of additional gates without using too few bits for the masking key k). For example, works presented in [4, 15] propose to use heuristics to reduce overhead.

3.2.4 Logic Locking

Logic locking allows the designer to insert *or*, *and*, *nor* or *nand* gates in the data path of the logic circuit of a Boolean function in order to lock the output to a fixed logic level (0 or 1) if the wrong unlocking key is applied. Let us consider that a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$ can be represented as a set of i Boolean subfunctions $\{f_0, f_1, \dots, f_{i-1}\}$. Logic locking of the Boolean function f by using the i-bit unlocking

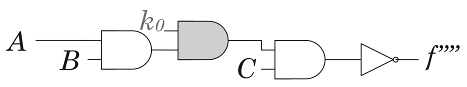


Fig. 3.4 Example of logic locking

word $k = \{k_0, k_1, \ldots, k_{i-1}\}$ is described by the following expression when f'''' is a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$ and $\odot$ is the *and* or *or* Boolean operator:

$$f''' = \{f_0 \odot_0 k_0, f_1 \odot_1 k_1, \ldots, f_{i-1} \odot_{i-1} k_{i-1}\}$$

$$\forall j \in [0, i-1] \begin{cases} \text{if } \odot_j \equiv \textit{and} \rightarrow k_j = 1 \rightarrow f_j \odot k_j = f_j \\ \text{if } \odot_j \equiv \textit{or} \rightarrow k_j = 0 \rightarrow f_j \odot k_j = f_j \end{cases} \tag{3.10}$$

The correct unlocking key k is found by using the laws in (3.10), and considering the type of inserted gate. As a didactic example, let us consider the following 3-input Boolean function $f\{0,1\}^3 \rightarrow \{0,1\}^1$:

$$f(A,B,C) = \overline{A.B.C}$$

This Boolean function could be expressed by the following expression:

$$\begin{cases} f(A,B,C) = f_1(f_0(A,B),C) \\ f_0(X,Y) = X.Y \\ f_1(X,Y) = \overline{X.Y} \end{cases}$$

A didactic example of logic locking of the Boolean function f is given in Fig. 3.4 where $\odot_0$ is an *and* gate. In this very simple example, only one gate is used to lock the logic behavior of the circuit. By following the laws in (3.10) we can determine the correct unlocking word $k = 1$ to obtain the correct behaviour. In Fig. 3.4 the additional locking gate is in gray.

Like for logic masking, the insertion of the locking gates has to be achieved without reducing performance and increasing area overhead. In the following section, we present a new method based on the graph analysis of an RTL netlist, which achieves efficient and secure logic locking.

Like in logic obfuscation and masking, it is possible to lock a circuit by acting on parts/levels other than the logic level. For example, recent works propose to lock the finite-state-machine [16, 17] or the input/output ports [18].

3.3 Proposed Graph Analysis-Based Logic Locking Scheme

As mentioned in Sect. 3.2.4, what we propose here is a new technique to select the nodes to include in the logic locking process. Indeed, since logic locking requires the insertion of extra logic gates, it is necessary to find the optimal place in the combinational netlist on which these extra gates should be inserted. According to the previously proposed definition, logic locking is the propagation of a fixed logic value from an internal node to one or several output(s). To achieve this, we need to identify sequences of gates that could propagate such a logic value. To this end, we represent the netlist as a graph. This representation is a convenient way of analyzing relations between logic gates and finding the optimal paths in a netlist that could propagate the logic locking value.

3.3.1 Implementation of Logic Locking

Before building the graph, we must identify the characteristics leading to the propagation of a locking value in a sequence of logic gates. First, it is worth noting that a specific *controlling value* exists for nonlinear logic gates. If this controlling value is applied to one of the logic gate's inputs, then the output is forced to a fixed, known value. For instance, setting one of the inputs of an *and* gate to 0 will set the output to 0. Table 3.2 summarizes the controlling values for the four 2-input nonlinear logic gates.

Next, for every node in the netlist, we define two values: V_{locks} and V_{forced}. V_{locks} is the controlling value of the gate that comes after this node. For instance, if a node is the input of an *or* gate, then $V_{locks} = 1$. V_{forced} is the value to which the node will be forced. For instance, if a node is the output of an *or* gate, $V_{forced} = 1$. It should be noted that in some cases $V_{locks} = \{0, 1\}$. This occurs if the node has a fan-out higher than one and spans gates with different controlling values. A node is useful for logic locking if it is forced to the controlling value of the following gate. Therefore, for sequences of nodes that can propagate a locking value, all the nodes meet the following criterion:

$$\textbf{Criterion 1}: V_{forced} \in V_{locks}$$

If Criterion 1 is verified for all the nodes in a sequence of nodes, then this sequence is able to propagate a locking value. In this case, forcing the first node to its controlling value will set all the nodes in the sequence at a fixed logic value. This is illustrated in Fig. 3.5.

Table 3.2 Controlling value and associated output value for 2-input nonlinear logic gates

Logic gate	Controlling value	Output value[a]
and	0	0
nand	0	1
or	1	1
nor	1	0

[a]when the controlling value is applied to one of the inputs

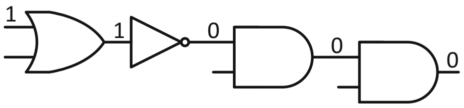


Fig. 3.5 Propagation of a locking value in a sequence of logic gates

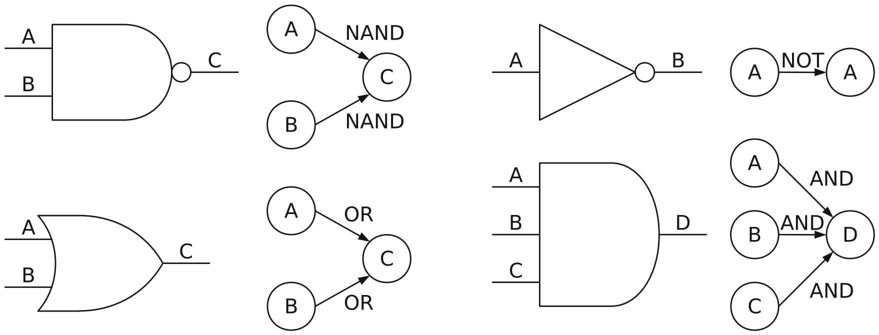


Fig. 3.6 Conversion from logic gates to graph elements

With this in mind, one can see how an output can be forced to a fixed logic value. By inserting logic gates at specific locations in the netlist, the designer will be able to set controlling values and force the outputs to a fixed value. The aim here is to select the most appropriate nodes, namely those at the beginning of sequences of gates like the one presented in Fig. 3.5. To achieve this aim, graph exploration techniques are used, and are presented in the following sections.

3.3.2 Graph Building

The original design file is an RTL description of the combinational netlist. The first step is to convert it into a directed acyclic graph. We chose to represent the netlist's nodes as vertices and the Boolean functions as edges. An example of conversion from logic gates to graph elements is shown in Fig. 3.6.

This is repeated for all logic gates of the netlist. A toy example of a netlist converted into a graph is shown in Fig. 3.7.

In order to identify which nodes satisfy criterion 1, V_{locks} and V_{forced} are computed for all the nodes in the netlist (i.e., all the vertices in the graph). This is done as follows: outgoing edges are used to compute V_{locks}, while incoming edges are used to compute V_{forced}. By convention, for the sake of the following computations, V_{locks} is set to $\{0, 1\}$ for the outputs. Table 3.3 shows V_{locks} and V_{forced} values computed for all the vertices of the graph shown in Fig. 3.7.

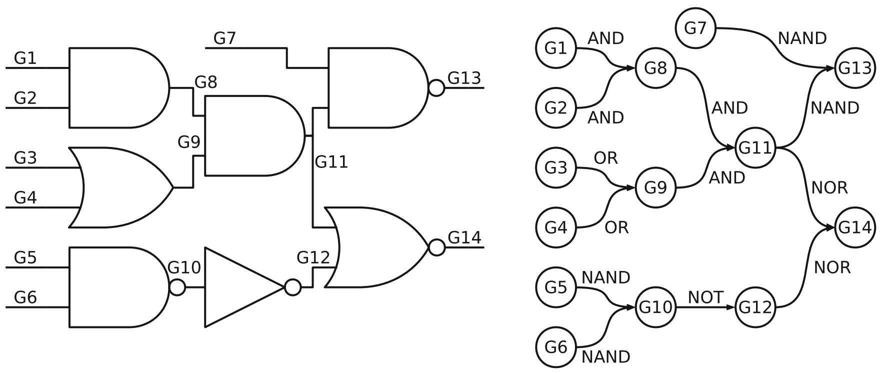


Fig. 3.7 Conversion from netlist to graph

Table 3.3 V_{locks} and V_{forced} values for all the nodes of the netlist shown in Fig. 3.7

Node	V_{forced}	V_{locks}	Node	V_{forced}	V_{locks}
G1	–	0	G8	0	0
G2	–	0	G9	1	0
G3	–	1	G10	1	0
G4	–	1	G11	0	$\{0, 1\}$
G5	–	0	G12	0	1
G6	–	0	G13	1	$\{0, 1\}$
G7	–	0	G14	0	$\{0, 1\}$

The next step is to identify which nodes cannot propagate the locking value. This means they do not fulfill criterion 1. If a node does not meet this criterion, its incoming edges are deleted. Thus in the previous example, incoming edges are deleted for G9, G10, and G12.

What is obtained at this stage is a highly disconnected graph, because the vast majority of vertices do not fulfill criterion 1. Since we want to achieve logic locking, connected components that do not contain any output must be removed from the graph. After applying this method to the graph in the previous example, we obtain the one shown in Fig. 3.8. The original netlist is shown too, and a path that can propagate a locking value is highlighted.

The final graph obtained at this stage comprises nodes that can all propagate a locking value to the output if they are forced to a specific logic value. Some of them, however, are better candidates, because they span a greater number of outputs or are more deeply integrated in the netlist. The selection algorithm used to identify the best nodes to act on is described in the following section.

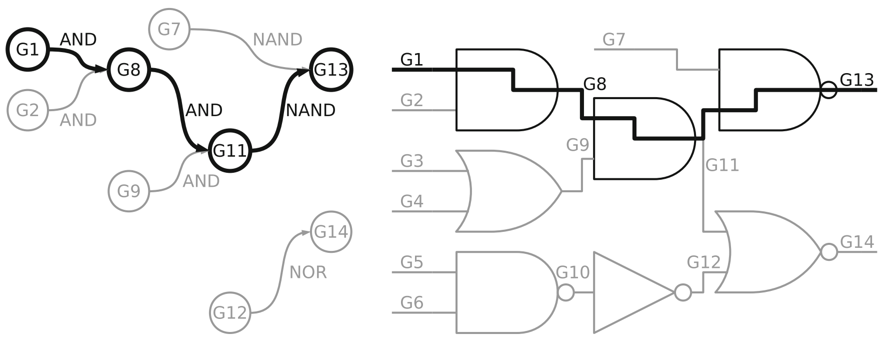


Fig. 3.8 Final graph and the original netlist showing a path that can propagate a locking value

3.3.3 Graph Analysis for Selection of Optimal Locking Nodes

At this stage, the graph is composed of several connected components. They all include at least one output, and are made of vertices that represent nodes able to propagate a locking value. These connected components can be classified in the four different categories depicted in Fig. 3.9.

In the first situation, shown in Fig. 3.9a, there is only one source vertex. Therefore, since the graph is directed, this vertex necessarily spans all the outputs, and can lock them all. It is consequently selected as the node to lock.

The second possibility, shown in Fig. 3.9b, occurs when a connected component comprises multiple source vertices but only one output. In order to embed the locking node as deeply as possible in the netlist, the distance between all source nodes and the output is computed. The furthest node from the output is selected as the node to lock.

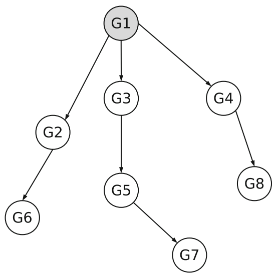
In the case depicted in Fig. 3.9c, there are multiple source vertices too. Some source vertices, however, do not span all the outputs. In order to lock as many outputs as possible with the smallest number of nodes to be modified, only the nodes spanning all the outputs are kept. If many nodes span all the outputs, then, as previously, the furthest one from the output is selected.

In the last situation, shown in Fig. 3.9d, multiple source vertices span multiple outputs, but none spans them all. The way to proceed here is to sort the source vertices according to the number of outputs they span. Next, they are greedily selected and added to the list of nodes to lock. This process is carried out until all the outputs are locked.

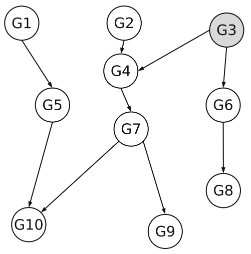
Note that the situations described above are sorted according to their computational complexity. The last case, which is the most computationally expensive, is also by far the least frequent.

One we have a list of nodes to modify, the last step is to add the extra locking gates that will be responsible for forcing these nodes to a specific value if the wrong key is applied.

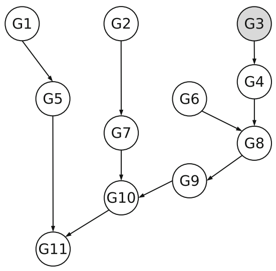
Fig. 3.9 **a** One source vertex, **b** Multiple source vertices, one output, **c** Multiple source vertices, multiple outputs, one (or more) source vertex spans all the outputs, **d** Multiple source vertices, multiple outputs, no vertex spanning all the outputs



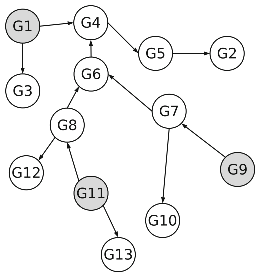
(a) One source vertex



(b) Multiple source vertices, one output



(c) Multiple source vertices, multiple outputs, one (or more) source vertex spans all the outputs



(d) Multiple source vertices, multiple outputs, no vertex spanning all the outputs

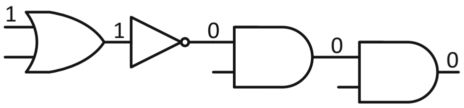


Fig. 3.10 Type of gate to insert according to the V_{forced} value and the associated unlocking bit

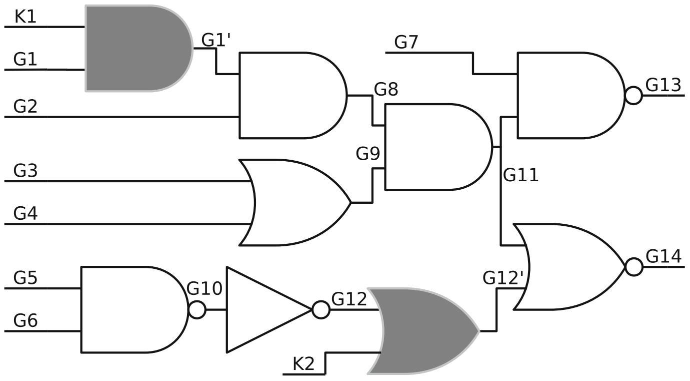


Fig. 3.11 Lockable netlist, inserted locking gates are in *gray*. The unlocking word is $(K_1K_2) =$ "10"

3.3.4 Netlist Modification

Now that we know which nodes to act on, the extra logic gates must be inserted. They will force these nodes to a specific value. The value to which each node must be forced is given by V_{locks}, the controlling value of the subsequent gate. If a node must be forced to 0, then an *and* gate is used. If a node must be forced to 1, then an *or* gate is used. This is shown in Fig. 3.10. The associated unlocking bit is the inverse of the controlling value of the inserted logic gate.

Coming back to the previous example, the nodes to be modified are G1 and G12. For G1, $V_{locks} = 0$ and for G12 $V_{locks} = 1$. Then the associated unlocking word (K_1, K_2) is "10." An *and* gate is used to force G1 to 0 if the wrong unlocking bit is applied, in this case: 0. An *or* gate is used to force G12 to 1 if the wrong unlocking bit is applied, in this case: 1. The final, lockable netlist is shown in Fig. 3.11.

3.4 Implementation Results

3.4.1 Logic Resources Overhead

The logic locking algorithm was implemented in Python, and makes use of the *igraph* module to handle graphs. We implemented the locking scheme on ITC'99

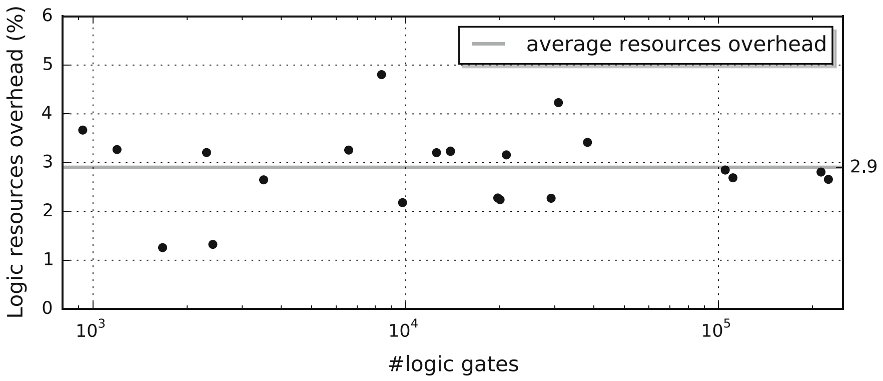


Fig. 3.12 Logic resources overhead obtained for logic locking

combinational benchmarks [19]. The netlists are described in VHDL. These benchmarks range from 1–225 k gates. The logic resources overhead is measured as the percentage of logic gates that must be added to the netlist in order to make it totally lockable. Results are shown in Fig. 3.12. The average resources overhead is 2.9 %. This is acceptable, and almost twice lower than the one authors obtained in [4]. Another interesting feature here is that the overhead remains approximately the same despite the increase in the number of gates in the original design. Protecting large netlists is consequently not more expensive than protecting smaller designs.

3.4.2 Analysis Time

Taking a step back, a major feature that will ensure the protection schemes are widely adopted is usability. It describes how easy it is for a designer to protect the IP core once it has been designed. In order to increase usability, a key point is the amount of time required to make the netlist lockable. Since these protection techniques could be integrated in EDA tools, the computation time should be reasonable. In Fig. 3.13, we provide a comparison of the computation time required to protect a netlist with both logic locking and logic masking methods. These results were obtained by executing the Python scripts on an Intel i5-4570 workstation, operating at 3.2 GHz and embedding 16Gb of RAM.

As can be seen in Fig. 3.13, the logic locking based method is more than ten thousand times faster than the method based on the logic masking. For instance, analyzing a 3,500-gate netlist requires four and a half hours with the method proposed in [4], whereas with the graph-analysis method, it takes less than one second. We extended our study to very large netlists of up to 225k gates. It turns out that the computation time increases quadratically. However, even for very large netlists, the computation

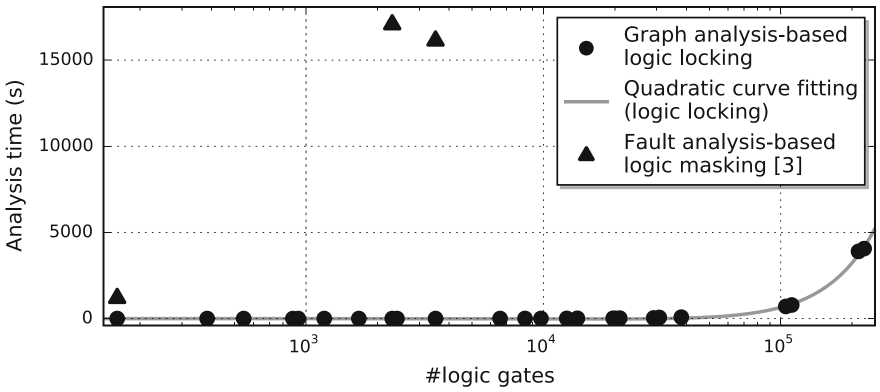


Fig. 3.13 Time required to analyze and modify the netlist

time is reasonable. For the largest one that includes 225k gates, slightly more than hour is required to make it lockable.

When it comes to the execution time, the main difference between the two protection methods is that the one proposed in [4] uses fault simulation to locate the nodes to modify. It relies on external tools that employ computationally heavy methods. Conversely, our protection technique is based on graphs, which are an effective way of representing netlists. In the context of EDA integration, our method is thus much more suitable and computationally more effective.

3.5 Evaluation

3.5.1 Correlation

In [4], the authors evaluate the efficiency of their locking scheme using the Hamming distance between the output of the original design and the output of the design when the wrong key is applied on the key inputs (i.e., when logic masking is activated). According to these authors, obtaining a 50 % Hamming distance on average is a proof that the protection scheme is efficient. However, we have shown in Sect. 3.2 that even simple circuits can exhibit such a characteristic, and that 50 % Hamming distance is simply one consequence of a zero correlation. We consequently use correlation to evaluate the efficiency of the protection scheme. The correlation is computed using Pearson's coefficient. The results are shown in Table 3.4. Since the standard deviation is zero when the outputs are locked by logic locking, Pearson's correlation coefficient is not defined. It can be considered as zero though, because when the output is locked, it provides no information about the normal behavior. Two methods are compared for logic masking: random and fault analysis-based node selection. Random selection

Table 3.4 Pearson's correlation coefficient computed for different node selection methods and key sizes

Benchmark	Key size	Logic masking		Logic locking
		Random [3]	Fault analysis [4]	Graph analysis
c432, 7 outputs, 189 nodes	32 bits	0.272	0.012	0
	64 bits	0.153	0.019	0
	128 bits	0.026	0.014	0
c5315, 123 outputs, 2362 nodes	32 bits	0.902	0.554	0
	64 bits	0.873	0.357	0
	128 bits	0.820	0.277	0
c7552, 108 outputs, 3612 nodes	32 bits	0.952	0.254	0
	64 bits	0.920	0.235	0
	128 bits	0.761	0.217	0

[3] rapidly becomes inefficient when the circuit's size increases. Randomly inserting 128 *xor* gates in a 3,612-node netlist only reduces the correlation to 0.761. Fault analysis-based logic masking is more efficient, and reduces the correlation faster as the key size increases. For large netlists, however, it fails to reduce it significantly. For example, the correlation only drops from 0.254 to 0.217 when the key size increases from 32 to 128 bits on C7552. For larger designs such as the ones considered in Sect. 3.4, the performance will probably be even worse.

We can conclude from this observation that correlation should not be used to evaluate a protection scheme. It is a cryptographic property, which should be only used in the appropriate frame. We give more details about security in Sect. 3.7. Instead of correlation, we propose a metric to evaluate protection schemes based on the insertion of extra logic gates, which is presented in the following subsection.

3.5.2 Logic Locking Metric

The intrinsic feature of a protection scheme based on the insertion of extra logic gates is altering the outputs using the extra gates. Therefore, two characteristics can be used to evaluate how effective these schemes are. The first one is: how many inputs are spanned by each extra logic gate? This is related to the amount of gates that have to be inserted to ensure total locking. If one gate locks multiple outputs, it is obviously more efficient than if multiple gates are required. The locking ratio is defined as follows:

$$Locking\ ratio = \frac{\#outputs}{\#locking\ gates} \tag{3.11}$$

Since the locking gates should be inserted as deeply as possible into the netlist, a second metric is: how far is the inserted gate from the outputs? The number of logic levels between the locking gate and the outputs is consequently also computed. The average distance between the inserted gates and the outputs is computed as the average number of logic levels on the shortest path between the inserted gates and every output that is reachable from them. The results we obtained when applying our graph-based insertion method for logic locking are presented in Table 3.5.

We can see that the number of outputs spanned by each locking gates is very close to 1. This basically means that, mostly, one logic locking gate is responsible for forcing one output. This is discussed in the following section. We can also see that the number of logic levels between the locking gates and the locked outputs is low. This could be a problem if the attacker has access to the RTL description of the design. Indeed, if the locking gates are located very close to the outputs, then the attacker can identify them easily and possibly modify the netlist to bypass the locking circuitry. This is why the locking gates need to be embedded as deeply as possible in the netlist.

Table 3.5 Evaluation of the proposed node selection technique by locking ratio and mean distance to outputs

Benchmark	#logic gates	Locking ratio	Average distance to outputs (logic levels)
c432	160	1.75	1.43
b10_C	172	1.13	1
b13_C	289	1.13	1.13
c880	383	1.63	3.39
b07_C	383	1.32	1.16
c1355	546	1.03	2
b04_C	652	1.02	1.11
b11_C	726	1.03	1.19
c1908	880	1.04	1
b05_C	927	1.82	1.52
b12_C	944	1.1	1.18
c2670	1193	1.68	2.38
c3540	1669	1.1	1.82
c5315	2307	1.68	2.07
c6288	2416	1.03	1
c7552	3512	1.16	1.5
b14_1_C	6569	1.15	1.48
b15_C	8367	1.12	1.69
b14_C	9767	1.16	1.42
b15_1_C	12543	1.12	2.06
b21_1_C	13898	1.14	1.33
b20_1_C	13899	1.14	1.32
b20_C	19682	1.15	1.36
b21_C	20027	1.14	1.29
b22_1_C	20983	1.14	1.35
b22_C	29162	1.15	1.36
b17_C	30777	1.11	1.76
b17_1_C	38116	1.11	1.97
b18_1_C	105102	1.12	1.74
b18_C	111241	1.12	1.74
	Average:	1.22	1.56

To this end, dummy logic levels can be inserted between the locking gate and the output, thereby achieving additional logic obfuscation as described in Sect. 3.2. For instance, an *or* gate can be replaced by the three gates depicted in Fig. 3.14. G is the node to be forced and K is the locking/unlocking input. Another node is picked randomly and used for the dummy logic. As depicted, the output value is

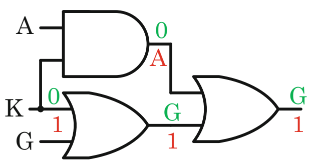


Fig. 3.14 *or* locking gate replacement with an extra logic level

either 1 or G, which means that locking is successful. Obviously, the increase in reverse engineering effort comes at the price of an increased area overhead. In order to add one logic level, three gates are inserted instead of one. If the designer wants to add a second dummy logic level, then the structure must be duplicated. Then five gates are inserted. The logic resources overhead is then $n(2k + 1)$, where n is the number of locking gates to be inserted and k is the number of dummy logic levels. In order to limit the overhead, dummy logic levels can be used only for the nodes that are too close to the outputs.

3.6 Security Analysis

3.6.1 Threat Model

To evaluate the security of logic locking, we must first distinguish the threat model of the actual context. Since we are trying to protect IP cores against illegal cloning, we must assume that the attacker has access to the original design, and can implement it. We make a stronger assumption by not limiting the number of implementations. Our aim for logic locking is only to make illegal copies nonfunctional. Thus, we first assume that the designer has access to the inputs used to unlock the design, i.e., the inputs to which the unlocking word encrypted with the secret key must be applied to unlock the circuit and to use it. In practical terms, the designer is able to write in a specific memory inside the chip, which will unlock the circuit if the correct value is provided. Moreover, since the designer appears to be legitimate at first sight, he also has access to test vectors.

3.6.2 Hill-Climbing Attack

Considering the threat model described above, a major concern expressed in [20] is the ease of a hill-climbing attack. It was described as an attack against the logic masking technique presented in [3]. However, it turns out to be equally efficient against logic locking. This is due to the tight link between the masking/locking inputs and the outputs. The attack procedure for logic masking described in [20] is as follows. First, pick a random key and apply it on the unlocking inputs. Compute the

Hamming distance between the actual and the expected output, given by the test vectors. Flip the first bit of the key. If the Hamming distance increases, then flip this bit again and repeat the action for all the bits of the key. Otherwise, if the Hamming distance decreases, move on to the next bit. The method is similar for logic locking, except that instead of using the Hamming distance as the function to minimize, the number of locked outputs is used. The main concern here is that, since there is a gradient toward the correct key in the key space, it can be easily recovered. In other words, the Hamming distance between the actual and expected output grows linearly with respect to the number of wrong key bits when logic masking is applied. Similarly, the number of outputs that are locked and the number of wrong key bits are correlated.

This is due to the fact that, as shown in Table 3.5, the ratio of the number of inserted gates to the number of outputs is close to one. In most cases, one gate is responsible for locking one output. This is a serious security concern. In this case, the security of the protection system is as low as the greatest number of key bits influencing one output. If the key bits and the outputs are connected pairwise, then the overall security level is 1 bit. In the following section, we discuss countermeasures against hill-climbing attacks.

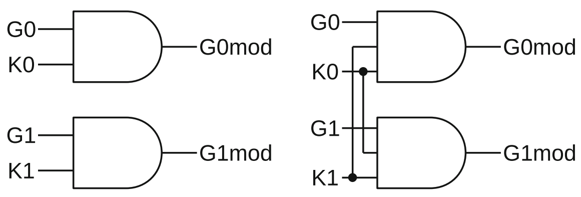
3.6.3 A Partial Countermeasure Against Hill-Climbing Attack

In order to avoid hill-climbing attacks, the correlation between the unlocking inputs and the outputs has to be reduced. One unlocking input should have an impact on multiple outputs, in order to hide the internal relation. Similarly, every output should be locked by several key inputs.

One possible countermeasure is to add some redundancy between the locking gates and the key inputs. This can be achieved by adding inputs to the locking gates. These inputs are connected to key inputs that have the same value as the first key input of the locking gate. For example, two locking gates for which the key bit is 1 can be associated, as depicted in Fig. 3.15. It follows that in order to obtain the correct values for $G0mod$ and $G1mod$, both $K0$ and $K1$ must have the correct value. It can be extended to add more key inputs to the locking gates, and more redundancy. However, this countermeasure is only partially effective. Indeed, it only increases the equivalent security level to the number of inputs added to the locking gates. Making it secure would require the locking gates to have a very large number of inputs, which is not feasible.

After another look at the previously described characteristic, it is very similar to the diffusion property of cryptographic functions. This led us to adopt another design plan for the protection scheme. Thus the logic locking module is only responsible for disturbing the original behavior. Security is ensured by using a separate cryptographic primitive. The overall architecture is described in the following section.

Fig. 3.15 Partial countermeasure against hill-climbing attack



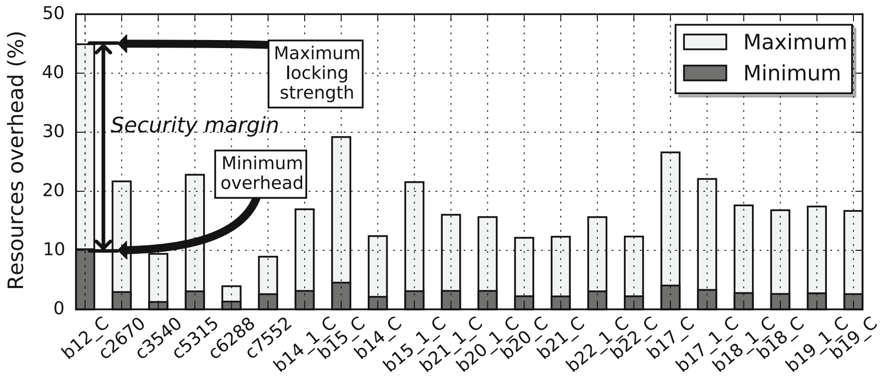
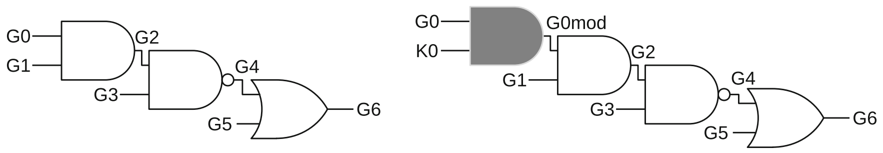


Fig. 3.16 Trade-off between locking strength and resources overhead

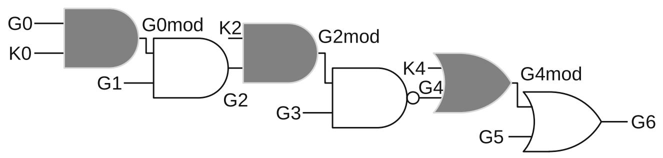
3.7 Architecture of a Complete Design Data Protection Scheme

3.7.1 Area/locking Strength Trade-Off

Before examining the whole protection scheme architecture, let us focus on the implementation of the logic locking module. After the graph has been built and analyzed, the final graph contains nodes that are all able to propagate a locking value. The method presented in Sect. 3.3.3 to select the best nodes to modify selects as few nodes as possible in the connected components to ensure total locking, but all the other nodes are also able to lock the associated output. Therefore, some extra locking gates can be added to increase the locking strength. Indeed, if the locking signal is carried by only one wire, it could be subject to side channel attacks such as optical injection [21] and its logic value can be flipped. In fact all the nodes found in the connected components of the final graph can be modified to increase the locking strength. This comes at the cost of increased logic resources overhead. This design trade-off is illustrated in Fig. 3.16, where the logic resources related to minimum overhead and maximum locking strength are given for all ISCAS'85 benchmarks. For b15_C for instance, the minimum overhead to achieve total functional locking is 4.52 %. However, up to 29 % extra resources can be added to further strengthen logic



(a) Original netlist and modified netlist with lowest area overhead



(b) Modified netlist with maximum locking strength and separated key bits

Fig. 3.17 **a** Original netlist and modified netlist with lowest area overhead. **b** Modified netlist with maximum locking strength and separated key bits

locking. The designer can decide on the acceptable resources overhead and increase the associated locking strength accordingly.

An example is given in Fig. 3.17. The original netlist and the netlist modified for logic locking with minimal overhead are shown in Fig. 3.17a. There is only one node forced, and one unlocking bit input. On the other hand, since all nodes $G0$, $G2$ and $G4$ can propagate a locking value, they can all be forced to increase the locking strength. This is shown in Fig. 3.17b. Three locking gates are inserted. The associated unlocking bits must all be set to their correct value in order to get the correct output. Of course, it comes at the price of an increased area overhead.

3.7.2 On the Need for a Cryptographic Primitive

In [4], the authors claim to achieve security by reaching 50 % Hamming distance between the original and masked outputs. Since in this case, security is not based on a cryptographic primitive, it is easily broken and [20] showed how it was possible to recover the key using a basic hill-climbing attack.

Only the system integrators allowed by the designer to unlock the IP core should be able to do so. If provable security is necessary, there is no other way than using a cryptographic primitive to obtain it. Another advantage is that such primitives, if chosen carefully, have been subject to a variety of attacks. Therefore, their security has been tested. The designer can then pick a strong cryptographic primitive that has successfully resisted multiple attacks, and implement it carefully. This will provide provable security of access to the normal operation of the IP core. For that reason, using a cryptographic primitive is necessary.

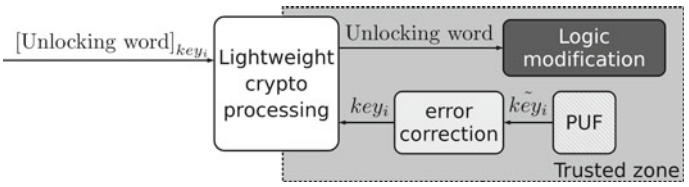


Fig. 3.18 Architecture of the proposed design protection module

3.7.3 Architecture

Owing to such considerations, we are now able to define the general architecture of the design protection scheme. It is depicted in Fig. 3.18.

The first block is the cryptographic primitive, which ensures secure access and avoids simple attacks. Using a lightweight, hardware-oriented algorithm is a good option here to limit the area overhead. The second block is a PUF, acting as a unique identifier, which is necessary in the case of IP distribution to uniquely identify all the instances of a particular design. It allows the designer to have a database containing all the IP core instances and their associated key. This is used to derive the key encrypting the unlocking word. In this way, it helps fulfill the following requirement: owning the key for one instance of the design should not help in unlocking another instance. Different types of PUFs are available, such as the TERO-PUF [22], the butterfly PUF [23] or the arbiter PUF [24]. It could also be achieved in the form of a secret word stored in nonvolatile memory. An error-correction module corrects the PUF's response. Finally, the unlocking word is deciphered and sent to the locking module. The locking module can implement logic encryption, masking or locking. Its role is to make the circuit unusable if the message sent to the cryptographic primitive is not the right unlocking word encrypted with the correct key associated with the circuit.

3.8 Summary

Design data protection schemes modifying the logic are a powerful way to render the circuit harder to reverse-engineer or unusable if it has been counterfeited. Several techniques to modify the logic are available, namely logic encryption, obfuscation, masking, or locking. They act on specific sites of the combinational logic part of the design. The method to select the sites to act on must be computationally efficient to be easily used, but also select the best sites. A graph analysis-based method is presented, which is fast and effective. Finally, we present design considerations, which include the integration of the logic modification in a wider protection scheme, in order to provide cryptographic strength and per-device uniqueness.

References

1. Y. Alkabani, F. Koushanfar, Active hardware metering for intellectual property protection and security, in *USENIX security*, USA, Boston, MA, Aug 2007, pp. 291–306
2. I. McLoughlin, Reverse engineering of embedded consumer electronic systems, in *IEEE 15th international symposium on consumer electronics*, Singapore, Singapore, June 2011, pp. 352–356
3. J.A. Roy, F. Koushanfar, I. Markov, EPIC: ending piracy of integrated circuits, in *Design, automation and test in Europe* (2008), pp. 1069–1074
4. J. Rajendran, H. Zhang, C. Zhang, G.S. Rose, Y. Pino, O. Sinanoglu, R. Karri, Fault analysis-based logic encryption. IEEE Trans. Comput. **64**(2), 410–424 (2015)
5. J. Rajendran, Y. Pino, O. Sinanoglu, R. Karri, Logic encryption: a fault analysis perspective, in *Design, automation & test in Europe conference*, Dresden, Germany, March 2012, pp. 953–958
6. S. Dupuis, P. Ba, G. Di Natale, M. Flottes, B. Rouzeyre, A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans, in *IEEE International On-Line Testing Symposium*, Girona, Spain, Platja d'Aro, June 2014, pp. 49–54
7. G. Hachez, A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards, Ph.D. dissertation, Université Catholique de Louvain, March 2003
8. J. Rajendran, Y. Pino, O. Sinanoglu, R. Karri, Security analysis of logic obfuscation, in *Annual design automation conference*, San Francisco CA, USA, June 2012, pp. 83–89
9. R.S. Chakraborty, S. Bhunia, HARPOON: an obfuscation-based SoC design methodology for hardware protection. IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst. **28**(10), 1493–1502 (2009)
10. R. Torrance, D. James, The state-of-the-art in semiconductor reverse engineering, in *Proceedings of the design automation conference, DAC 2011*, San Diego, California, USA, 5–10 Jun 2011, 2011, pp. 333–338
11. M. Brzozowski, V.N. Yarmolik, Obfuscation as intellectual rights protection in VHDL language, in *International conference on computer information systems and industrial management applications*, IEEE Computer Society, Elk, Poland, Jun 2007, pp. 337–340
12. U. Meyer-Baese, E. Castillo, G. Botella, L. Parrilla, A. Garca, Intellectual property protection (IPP) using obfuscation in C, VHDL, and verilog coding, in *SPIE defense, security, and sensing*, Orlando, Florida, USA, Jun 2011
13. J. Rajendran, M. Sam, O. Sinanoglu, R. Karri, Security analysis of integrated circuit camouflaging, in *ACM conference on computer & communications security*, Berlin, Germany, Nov 2013, pp. 709–720
14. SypherMedia, Circuit camouflage technology (2012)
15. A. Baumgarten, A. Tyagi, J. Zambreno, Preventing IC piracy using reconfigurable logic barriers. IEEE Des. Test Comput. **27**(1), 66–75 (2010)
16. E. Jung, C. Hung, M. Yang, S. Choi, An locking and unlocking primitive function of FSM-modeled sequential systems based on extracting logical property. Int. J. Inf. **16**(8), 6279–6290 (2013)
17. M.T. Rahman, D. Forte, Q. Shi, G.K. Contreras, M.M. Tehranipoor, CSST: preventing distribution of unlicensed and rejected ICs by untrusted foundry and assembly, in *IEEE international symposium on defect and fault tolerance in VLSI and nanotechnology systems*, Netherlands, Amsterdam, Oct 2014, pp. 46–51
18. A. Basak, Y. Zheng, S. Bhunia, Active defense against counterfeiting attacks through robust antifuse-based on-chip locks, in *IEEE 32nd VLSI test symposium*, USA, Napa CA, Apr 2014, pp. 1–6
19. S. Davidson, ITC'99 benchmark circuits—preliminary results, in *IEEE international test conference*, NJ, USA, Atlantic City, Sept 1999, p. 1125
20. S.M. Plaza, I.L. Markov, Protecting integrated circuits from piracy with test-aware logic locking, in *International conference on computer aided design*, San Jose, CA, USA, Nov 2014

21. S.P. Skorobogatov, R.J. Anderson, Optical fault induction attacks, in *International workshop on cryptographic hardware and embedded systems*, San Fransisco CA, USA, Aug 2002
22. L. Bossuet, X.T. Ngo, Z. Cherif, V. Fischer, A PUF based on transient effect ring oscillator and insensitive to locking phenomenon. IEEE Trans. Emerg. Top. Comput. **2**(1), 30–36 (2014)
23. S.S. Kumar, J. Guajardo, R. Maes, G.J. Schrijen, P. Tuyls, The butterfly PUF protecting IP on every FPGA, in *IEEE international workshop on hardware-oriented security and trust*, USA, Anaheim CA, Jun 2008, pp. 67–70
24. J.W. Lee, D. Lim, B. Gassend, G.E. Suh, M. van Disk, S. Devadas, A technique to build a secret key in integrated circuits for identification and authentication applications, in *Symposium on VLSI circuits*, Jun 2004, pp. 176–179