From Secured Logic to IP Protection

Brice Colombier*, Lilian Bossuet, David Hély

Laboratoire Hubert Curien, University of Lyon, 42000, Saint-Etienne, France
LCIS Grenoble Institute of Technology, 26000, Valence, France
*corresponding author contact: b.colombier@univ-st-etienne.fr +334 77 91 57 92

Abstract — **Design and reuse has become a very common practice in the electronics design industry. IP cores are easily sold by designers to system integrators. However, several cases of counterfeiting and illegal copying have been reported and design protection techniques have been developed in response. Among these techniques, we focus on modifications at logic level aimed at active design protection. This is the first paper to provide a formal description and definition of the following techniques used to protect integrated circuits and IP cores against theft, counterfeiting, cloning and illegal copy: logic encryption, logic obfuscation, logic masking, and logic locking. In the second part of the paper, we present a new technique to insert gates in the data path of a logic circuit in order to lock it. Based on graph analysis, this method involves low overhead implementation and is more than ten thousand times faster than former fault analysis-based logic masking techniques when it comes to selecting the nodes to modify. Finally, we discuss the design requirements of a strong design protection scheme.**

Index Terms—**Intellectual property protection, logic encryption, logic obfuscation, logic functional locking.**

I. Introduction

PROTECTION of the intellectual property of IP core designers is a hot topic and remains an open question. Because of the ever increasing complexity of electronic systems, full in-house design is no longer the norm. IP core designers provide system integrators with a wide IP portfolio in which they can select the functional block best suited to their needs. EDA companies offer IP cores that are accessible directly from their design software. Online marketplaces are expanding, and system integrators can use them to compare a whole range of different IP cores, and choose the most appropriate for their system. This paradigm helps reduce time-to-market because the IP cores have been thoroughly tested, are “silicon-proven” and can be integrated all together in a more complex system. From the integrator’s point of view, this is a great help. The IP core designer, however, is faced with a problem: since the IP core is made available as a data file, how can the designer control how many times the IP core is actually instantiated? To cut a long story short, how can overusing be prevented? In addition, another problem is counterfeiting, which is not specific to IP cores, but also occurs with standard integrated circuits. Illegal copying, cloning and theft are long known threats to design data.

Logic protection schemes have been developed to counter these risks. They can be classified as *passive* and *active* protection schemes. Passive protection detects but does not prevent an illegal action, whereas active protection techniques make the illegal action much harder to carry out or even pointless. Both types of protection schemes can act at different design levels. Here, we focus on logic protection schemes. They consist in modifying the RTL description of the design, and adding extra elements to protect it. However, there is currently a lack of formal classification of logic protection schemes.

Once a protection scheme is selected by the designer, the second point to be addressed is the method used to select which parts of the design should be modified. Here we propose a new method, based on graph analysis, to select the nodes of a netlist to modify to achieve logic locking. We compare our method with the state-of-the-art selection technique used for logic masking proposed in [1], which uses fault-analysis techniques.

The remainder of this article is organized as follows. In section II, we provide a formal framework for logic protection schemes by defining *logic encryption*, *logic obfuscation*, *logic masking* and *logic locking* and give examples of techniques for each. In section III, we present a new graph-based algorithm that selects the optimal nodes to be modified to achieve logic locking of a combinational netlist. In section IV, we present the results of implementation, specifically the logic resources overhead and analysis time. In section V, we evaluate the proposed method and develop associated metrics. In section VI we describe a threat model and perform a security analysis of the protection schemes considered. In section VII, 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.

II. 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 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 paper 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 sub-sections, the original (not protected) n-input, l-output logic function is formalized by a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$.

A. Logic encryption

The term "logic encryption" is used when a specific symmetric encryption function ξ_f of $GF(2^l)$ is applied to f. Formally, it's 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:

$$\psi_f(f') = \psi_f(\xi_f(f,k),k) = f \qquad (1)$$

Functions ξ_f and ψ_f must meet the following requirements:

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

$$\xi_f(f,k_i) \neq \xi_f(f,k_j) \qquad (2)$$

$$\psi_f(\xi_f(f,k_i),k_i) \neq (\xi_f(f,k_i),k_j) \qquad (3)$$

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

$$\forall\, k \in \{0,1\}^m \qquad Corr(\xi_f(f,k),f) \cong 0 \qquad (4)$$

$$\forall\, k \in \{0,1\}^m \quad Corr\left(\psi_f(\xi_f(f,k),k),\xi_f(f,k)\right) \cong 0 \qquad (5)$$

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:

$$\forall\, k \in \{0,1\}^m \qquad \frac{\sum HD(\xi_f(f\{0,1\}^n,k),f\{0,1\}^n)}{2^n-1} \cong 50\% \qquad (6)$$

$$\forall\, k \in \{0,1\}^m \quad \frac{\sum HD\left(\psi_f(\xi_f(f\{0,1\}^n,k)),\xi_f(f\{0,1\}^n,k)\right)}{2^n-1} \cong 50\% \qquad (7)$$

Some works [1,2,3] 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 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 (6) is equal to 50% but the correlation defined in (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 original design*" [2]. With this definition, logic encryption does not respect the expressions (1) to (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 $f\ \{0,1\}^3 \rightarrow \{0,1\}^1$:

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

Figure 1 is a schematic diagram of the encrypted logic circuit. This includes the original logic circuit that computes the Boolean function f, the encryption function ξ_f that computes the encrypted Boolean function f' using an embedded secret key k, and the decryption function ψ_f that outputs the correct result of the Boolean function f if and only if the correct key k is applied on the external key input.

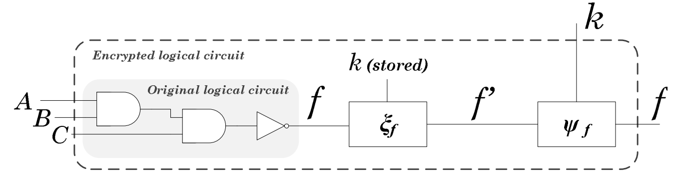


Fig. 1. Example of logic encryption

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. Now, an efficient symmetric encryption has to use at least a 128-bit key. All protection schemes that include a secret key that has only a few bits (3, 5, 10 etc.) fail to provide the designer with any security because of the feasibility of a brute force attack.

B. 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 [4]: "*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 $f''\{0,1\}^n \to \{0,1\}^l$. The function γ_f must meet the following requirement for any input $x \in \{0,1\}^l$:

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

$$\forall x \in \{0,1\} \qquad f''(x) = f(x) \quad (8)$$

Some works present logic obfuscation but do not fulfill the requirement (8) [5,6]. Most of these works use a secret key that changes the behavior of the original logic function. Although authors refer it as “functional obfuscation”, these works are typical cases of logic masking.

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 [7]. 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 *minterm* canonical form) in which the Boolean function is represented and implemented as a sum of *minterms*.

As an example, let us consider the following 3-input Boolean function $f\ \{0,1\}^3 \to \{0,1\}^1$:

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

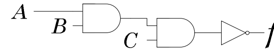
The obfuscation strategy we use here is for didactic purpose only. We only give a simple example in order to demonstrate the proportional increase of reverse-engineering time and area overhead.

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

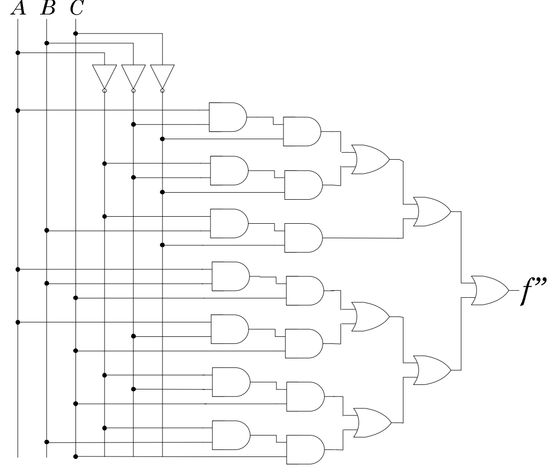
$$f''(A,B,C) = A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot \overline{C} + A \cdot B \cdot C + A \cdot \overline{B} \cdot C + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot C$$

f and *f’’* follow requirement (8). Figures 2-a and 2-b show the logic schematics of the two functions with only 2-input AND and OR gates and invertors (other types of gates could also be used).

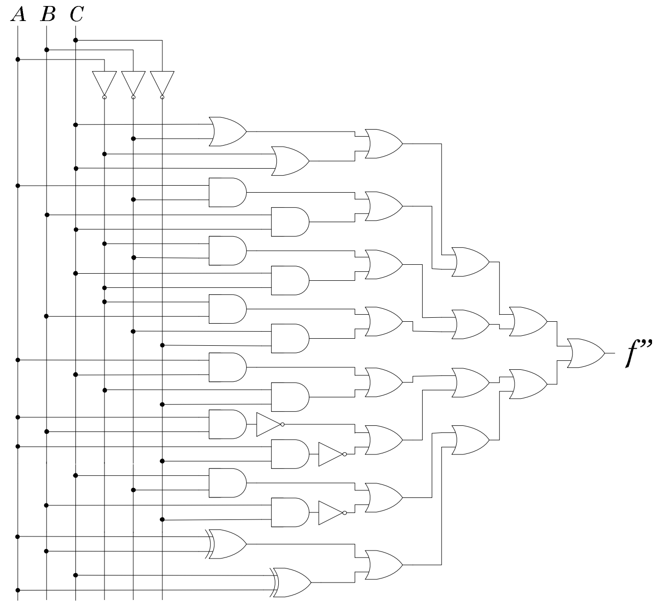
In order to obscure a Boolean function, the designer can apply to *f’’* some of the Boolean logic laws (absorption, complementary, common identities, etc.) and DeMorgan’s theorem to increase the number of gates used in the hardware implementation. For example, by also using some redundant logic operations,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



(a) Original Boolean function implementation



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



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

Fig. 2. Logic circuits to implement Boolean functions *f* (a), *f”* after one step of obfuscation (b) and *f”* after two steps of obfuscation (c).

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

Again, *f* and *f''* meet requirement (5). Figure 1-c shows the logic schematic of *f''* after this second step of obfuscation. The designer can also insert dummy logic to further increase the reverse engineering effort.

Table I shows the logic resources required for each logic circuit in figure 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 2-c is 14.58 times greater than the time required to reverse engineer the original circuit.

TABLE I
LOGIC RESOURCE REQUIREMENTS AND TIMING OVERHEAD FOR REVERSE ENGINEERING OF THE CIRCUITS DESCRIBED IN FIGURE 1

Boolean function	*Logic circuit*	*Logic gate requirement*				*Gate Equivalent*	*Area / Reverse engineering time overhead*
		INV	*AND*	*OR*	*XOR*		
f	*1-a*	*1*	*2*			*4.01*	*-*
f' after the first step of obfuscation	*1-b*	*3*	*14*	*6*		*35.41*	*+ 883 %*
f' after the second step of obfuscation	*1-c*	*6*	*12*	*17*	*2*	*58.47*	*+ 1 458 %*

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 level of HDL [8,9] and at the level of the layout [10,11].

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.

C. 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 [12]. Let us consider that a Boolean function $f\{0,1\}^n \rightarrow \{0,1\}^l$ could be represented as a set of *i* Boolean sub-functions $\{f_0, f_1, \ldots, f_{i-1}\}$. Logic masking of the Boolean function *f* by using the i-bit masking key $k=\{k_0, k_1, \ldots, k_{i-1}\}$ is described by the following expression, where *f'''* is a Boolean function $\{0,1\}^n \rightarrow \{0,1\}^l$ and $\ominus$ is the *xor* or *xnor* Boolean operator:

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

$$\forall j \in \{0, i-1\} \begin{cases} if\ \oint_j \equiv xor \Rightarrow k_j = 1 \Rightarrow f_j \oint k_j = f_j \\ if\ \oint_j \equiv nxor \Rightarrow k_j = 0 \Rightarrow f_j \oint k_j = f_j \end{cases} \quad (9)$$

The correct masking key *k* is found by using the laws in (9), and considering the type of inserted gate.

As a didactic example, let us consider the following 3-input Boolean function $f\ \{0,1\}^3 \rightarrow \{0,1\}^1$:

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

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

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

A didactic example of logic masking of the Boolean function *f* is given in figure 3, where $\ominus_0$ is an *xnor* gate and $\ominus_1$ is a *xor* gate. According to the laws in (9), we can determine the correct masking k={0,1} needed to obtain the original logic behaviour. In figure 3, additional masking gates are in grey.

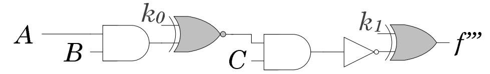


Fig. 3. Example of logic masking

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 [13] and [1] propose to use heuristics to reduce overhead.

D. Logic locking

Logic locking allows the designer to insert *or* or *and* 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 sub-functions $\{f_0, f_1, \ldots, f_{i-1}\}$. Logic locking of the Boolean function *f* by using the i-bit unlocking key $k = \{k_0, k_1, \ldots, k_{i-1}\}$ is described by the following expression when *f''''* is a Boolean function $\{0,1\}^n \rightarrow \{0,1\}^l$ and $\odot$ is the and or *or* Boolean operator:

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

$$\forall j \in \{0, i-1\} \begin{cases} if\ \odot_j \equiv and \Rightarrow k_j = 1 \Rightarrow f_j \odot k_j = f_j \\ if\ \odot_j \equiv or \Rightarrow k_j = 0 \Rightarrow f_j \odot k_j = f_j \end{cases} \quad (10)$$

The correct unlocking key k is found by using the laws in (10), depending on the type of inserted gate.

As a didactic example, let us consider the following 3-input Boolean function $f\ \{0,1\}^3 \rightarrow \{0,1\}^1$:

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

This Boolean function could be expressed by the following expression:

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

A didactic example of logic locking of the Boolean f is given in figure 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 (10), we can determine the correct masking k=1 to obtain the correct behavior. In figure 4, the additional locking gate is in grey.

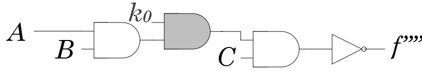


Fig. 4. Example of logic locking

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 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 [14,15] or the input/output ports [16].

III. Proposed Graph Analysis-Based Logic Locking Scheme

As mentioned in section II-d), 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 spots in the combinational netlist on which these extra gates should be inserted. According to the previously proposed definition, logic locking can be 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.

A. 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 non-linear 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 II summarizes the controlling values for the four 2-input non-linear logic gates.

TABLE II
CONTROLLING VALUE AND THE ASSOCIATED OUTPUT VALUE FOR ALL 2-INPUT NON-LINEAR LOGIC GATES

Logic gate	Controlling value	Output value[1]
AND	0	0
NAND	0	1
OR	1	1
NOR	1	0

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

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, then $V_{forced} = 1$. It should be noted that, sometimes, $V_{locks} = \{0,1\}$, 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:

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 figure 5. Here, forcing one of the inputs of the first *or* gate to a 1 logic value forces the output of the *and* gate on the right to 0.

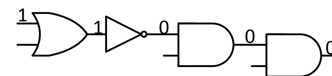


Fig. 5. Propagation of a locking value in a sequence of logic gates

With this in mind, one can see how an output can easily be forced to a fixed logic value. By inserting logic gates at specific locations in the netlist, the designer will be able to force the outputs to a fixed value by controlling the value of specific internal nodes. The aim here is to select the most appropriate nodes, namely those at the beginning of sequences of gates like the one presented in figure 6. To achieve this aim, graph exploration techniques are used, and are presented in the following sections.

B. 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 figure 6.

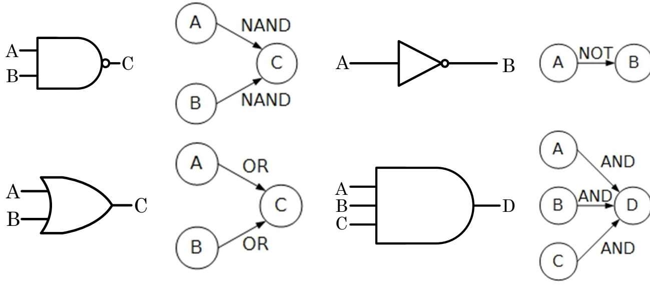


Fig. 6. Conversion from logic gates to graph elements

This is repeated for all logic gates of the netlist. A toy example of a netlist converted into a graph is shown in figure 7:

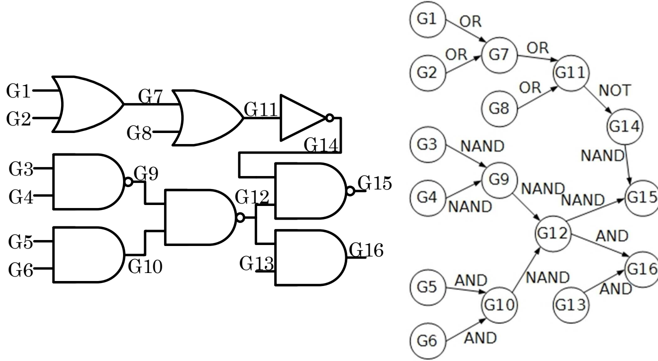


Fig. 7. Conversion from netlist to graph

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 III shows V_{locks} and V_{forced} values computed for all the vertices of the graph shown in figure 7.

TABLE III
V_{locks} AND V_{forced} VALUES FOR ALL THE NODES OF THE NETLIST SHOWN IN FIGURE 7

Node	V_{forced}	V_{locks}	Node	V_{forced}	V_{locks}
G1	-	1	G9	1	0
G2	-	1	G10	0	0
G3	-	0	G11	1	1
G4	-	0	G12	1	0
G5	-	0	G13	-	0
G6	-	0	G14	0	0
G7	1	1	G15	1	{0, 1}
G8	-	1	G16	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 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 figure 8. The original netlist is disregarded, and a path that can propagate a locking value is highlighted.

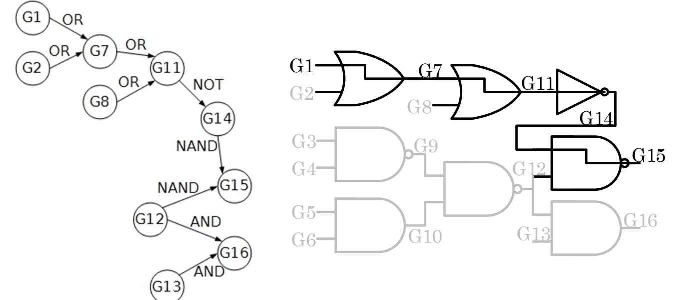


Fig. 8. Final graph and the original netlist showing a path that can propagate a locking value

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 larger number of outputs or are more deeply integrated in the netlist. The selection algorithm we used to identify the best nodes to act on is described in the following section.

C. 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 up of vertices that represent nodes able to propagate a locking value. These connected components can be classified in the four different categories depicted in figure 9.

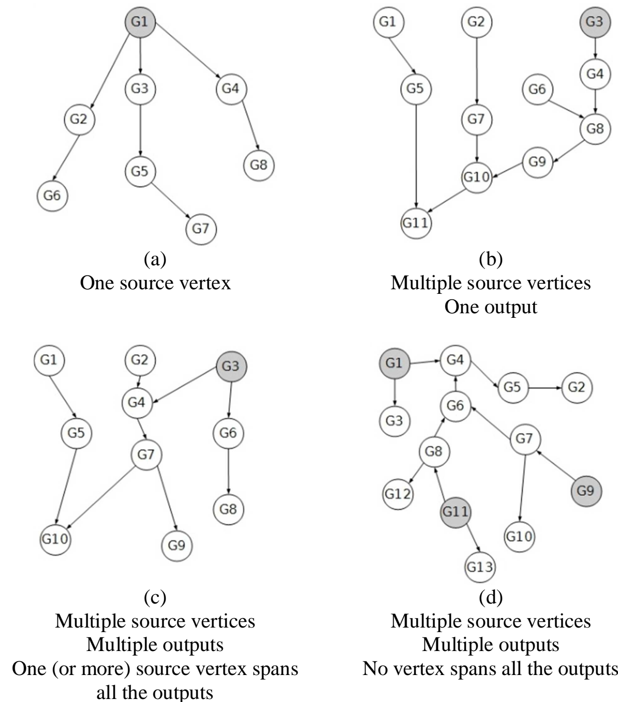


Fig. 9. Different types of connected components found in the final graph

In the first situation, shown in figure 9.a, there is only one source vertex. Therefore, since the graph is directed, it necessarily spans all the outputs, and can lock them all. It is consequently selected as the node to lock.
The second possibility, shown in figure 9.b, 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 figure 9.c, 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 retained. If many nodes span all the outputs, then, as previously, the one furthest from the output is selected.
In the last situation, shown in figure 9.d, 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.
The node selection process is summarized in figure 10.

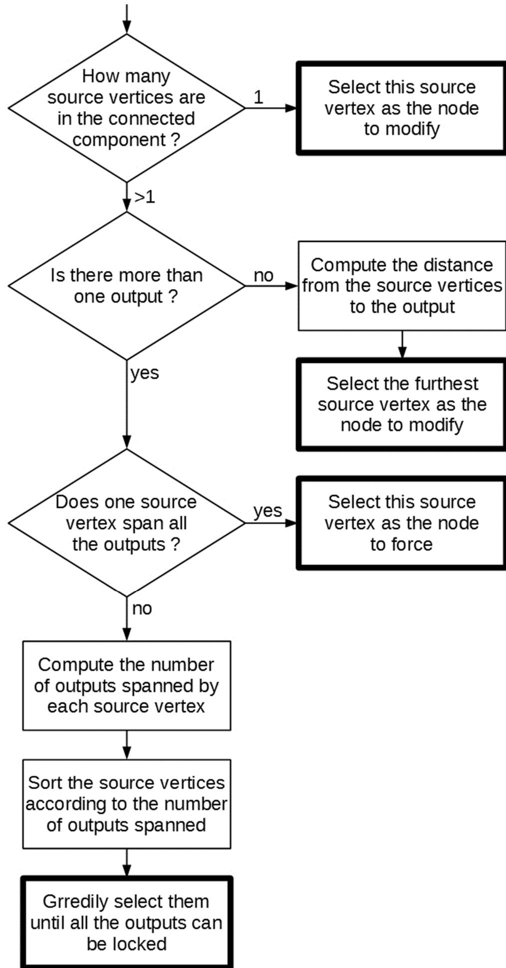


Fig. 10. Flowchart of the node selection process

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.

D. *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}. If a node must be forced to 0, then an *and* gate is used. If a node must be forced to 1, than an *or* gate is used. The associated key-bit is the inverse of the controlling value of the inserted logic gate. This is shown in figure 11.

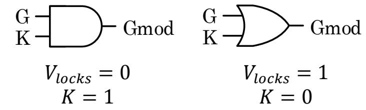


Fig. 11. Type of gate to insert according to V_{forced} value

Coming back to the previous example, the nodes to be modified are G1 and G13. For G1, $V_{locks} = 1$ and for G13, $V_{locks} = 0$. Then the associated unlocking key ($K_1 K_0$) is 01. An *or* gate is used to force G1 to 1 if the wrong key bit is applied, in this case: 1. An *and* gate is used to force G13 to 0 if the wrong key bit is applied, in this case: 0. The final, lockable netlist is shown in figure 12:

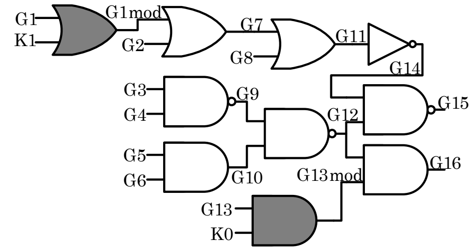


Fig. 12. Lockable netlist, locking gates are in dark grey.
The unlocking word is: $K_1 K_0 = 01$

IV. Implementation Results

A. *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 combinational benchmarks [17]. The netlists are described in VHDL. These benchmarks range from 1 k to 225 k gates. They are good reference designs. 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 figure 13. The average resources overhead is 2.9%. This is acceptable, and almost twice less than the one authors obtained in [1]. Another interesting feature here is that the overhead remains approximately the same despite the increase in the number of gates. Protecting large netlists is

consequently no more expensive than protecting smaller designs.

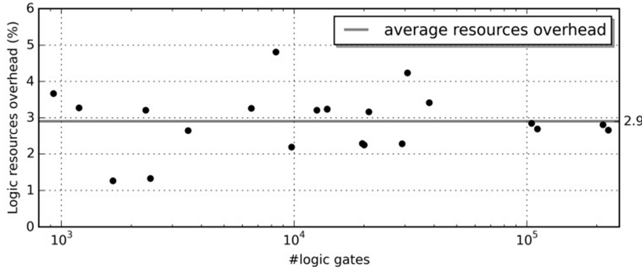


Fig. 13. Logic resources overhead obtained for functional locking

B. *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 figure 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 3.2GHz and embedding 16Gb of RAM.

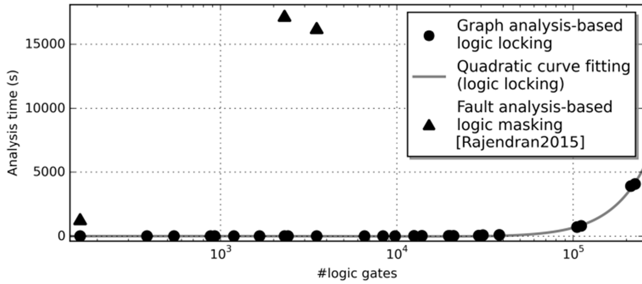


Fig. 14. Time required to analyze and modify the netlist

As can be seen in figure 14, the logic locking based method is more than ten thousand times faster than the method based on logic masking. For instance, analyzing a 3,500-gate netlist requires four and a half hours with the method proposed in [1], whereas with the method we propose, it takes less than one second. We extended our study to very large netlists of up to 225 k gates. It turns out that the computation time increases quadratically. However, even for very large netlists, the computation time is reasonable. For the largest one that includes 225 k 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 method proposed in [1] 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.

C. *Impact on the critical path delay*

The impact of the locking gates insertion on the critical path is minimal. First of all, non-linear logic gates have a lower propagation time than *xor* gates for example. Moreover, the sequences that can propagate a locking value might not be on the critical path. Thus the critical path will not remain in the final graph and no extra gate will be inserted in it. In case the critical path can propagate a locking value, it will require at most one extra gate to lock it. Thus the impact on the critical path delay will be negligible.

V. EVALUATION

A. *Correlation*

In [1], the authors evaluate the efficiency of their locking scheme using the Hamming distance between the output of the original device and the output of the device 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 proof that the protection scheme is efficient. However, we have shown in section II 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 IV. 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 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 [12] rapidly becomes inefficient with an increase in the size of the circuits. 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 goes 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 section IV, the performance will probably be even worse.

TABLE IV
PEARSON'S CORRELATION COEFFICIENT COMPUTED FOR DIFFERENT NODE SELECTION METHODS AND KEY SIZES

		Logic masking		Logic locking
Benchmark	Key size	Random [12]	Fault analysis [1]	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

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 section VII below. Instead of a correlation, we developed a metric to evaluate protection schemes based on the insertion of extra logic gates, which is presented in the following subsection.

B. *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 functional 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:

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

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 V.

TABLE V
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

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. To this end, dummy logic levels can be inserted between the locking gate and the output, thereby achieving logic obfuscation as described in section II. For instance, an *or* gate can be replaced by the three gates depicted in figure 15. 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 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 will have to be duplicated. Then five gates are inserted. The logic resources overhead is $k * (2n + 1)$, where k is the number of locking gates to be inserted and n 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.

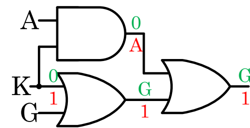


Fig. 15. OR locking gate replacement with an extra logic level

VI. SECURITY ANALYSIS

A. *Threat model*

To evaluate the security of logic locking, we must first distinguish the threat model from 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 non-functional. Thus we first assume that the designer has access to the unlocking inputs, i.e. the inputs to which the 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.

B. *Hill-climbing attack*

Considering the threat model described above, a major concern expressed in [18] is the ease of a hill-climbing attack. It was described as an attack against the logic masking technique presented in [12]. 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 [18] 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 towards 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 V, 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.

C. *A 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 figure 16. 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.

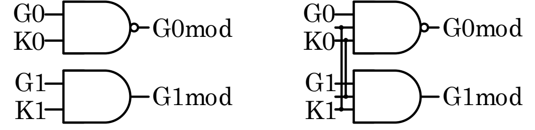


Fig. 16. Partial countermeasure against hill-climbing attack

When we had another look at the previously described characteristic we realized 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

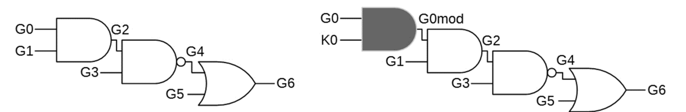
VII. Architecture of a Complete Design Data Protection Scheme

A. *Area/locking strength tradeoff*

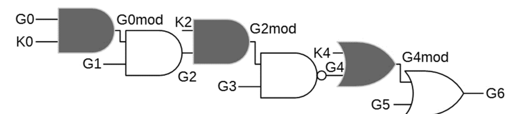
In order to increase the security of the logic locking scheme, multiple locking gates could be inserted to force only one internal node. Such a redundant locking strategy strengthens the security of the system. Before examining the whole protection scheme architecture, let us focus on the implementation 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 section III.C 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. This is presented in figure 17.

Figure 17-a shows the original netlist and the modified one. In the modified version, only one gate is inserted for minimal overhead. The locking strength is low, wince the locking value is set by one gate only.

In figure 17-b, the locking value is set by three gates. Therefore, all three gates should be unlocked to unlock the associated output. Thus the locking strength is higher, but so is the area overhead.



(a) Original netlist and modified netlist with only one locking gate.



(a) Modified netlist with three gates forcing the locking value.

Fig. 17. Insertion of one or multiple locking gates to lock an output

If the locking signal is carried by only one wire, it could be subject to side channel attacks such as optical injection [19] 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 figure 18, 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 locking. The designer can decide on the acceptable resources overhead and increase the associated locking strength accordingly.

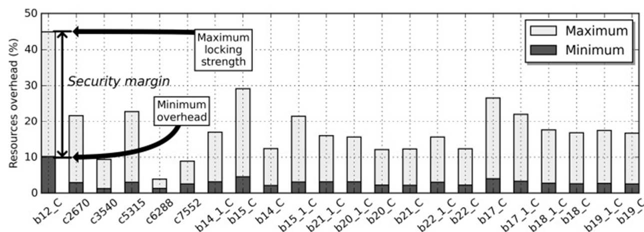


Fig. 18. Trade-off between locking strength and resources overhead

B. *On the need for a cryptographic primitive*

In [1], 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 [18] 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 him/her with provable security of access to the normal behavior of the IP core. For that reason, using a cryptographic primitive is necessary.

C. *Architecture*

Owing to such considerations, we are now able to define the general architecture of the design protection scheme. It is composed of three main blocks, shown in figure 19.

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 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 identifier could also be used to derive the unlocking key. 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. One possible implementation of a unique identifier is a PUF [20]. It could also be achieved in the form of a secret word stored in non-volatile memory.

PUFs have already been used in previous works, known as metering [21]. However, those works make the assumption that PUF responses are perfectly stable, and use the PUF response directly to mask internal nodes. Unfortunately, several experimental studies show that it is hard to obtain perfectly stable PUF responses which could be used as a key directly without an expensive error correction. Conversely, we take the instability of PUF responses into account and use the PUF for identification only.

The final block is the locking module. Its role is to make the circuit unusable if the wrong key is sent to the cryptographic primitive.

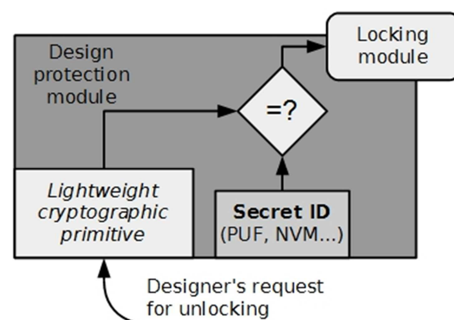


Fig. 19. Architecture of the proposed design protection module

VIII. CONCLUSION

In the first part of this paper, we provided precise definitions of design protection schemes involving modifications of the logic. This is necessary when developing new protection schemes, since all have several pros and cons. Next, we proposed a new method to select the nodes to be modified for logic locking. Based on graph analysis, this method was shown to be effective, in terms of both induced area overhead and computational complexity. A comparison with state-of-the-art fault analysis-based logic masking techniques was then performed to emphasize these advantages. The last two sections provided insight into the security level of both protection schemes. We highlighted the fact that existing logic-based protection techniques cannot be considered as secure since they are subject to very simple attacks. Finally, we proposed an overall architecture for a robust complete protection scheme, embedding a locking module to disable the functionality, a cryptographic primitive to provide provable security and a unique identifier that allows precise metering.

ACKNOWLEDGEMENTS

The work presented in this paper was conducted in the frame of the SALWARE project number ANR-13-JS03-0003 supported by the French "*Agence Nationale de la Recherche*" and by the French "*Fondation de Recherche pour l'Aéronautique et l'Espace*", funding for this project was also provided by a grant from la région Rhône-Alpes.

REFERENCES

[1] J. Rajendran, H. Zhang, C. Zhang, G.S. Rose, Y. Pino, O. Sinanoglu, R. Karri, "Fault Analysis-Based Logic Encryption," *IEEE Transactions on Computers*, vol.64, no.2, pp.410,424, Feb. 2015

[2] J Rajendran, Y. Pino, O. Sinanoglu, R. Karri, "Logic encryption: A fault analysis perspective," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012* , pp.953,958, 12-16 March 2012

[3] S. Dupuis, P.S. Ba, G. Di Natale, M.L. Flottes, B. Rouzeyre, "A Novel Hardware Logic Encryption Technique for thwarting Illegal Overproduction and Hardware Trojans", *20th IEEE International On-Line Testing Symposium, IOLTS'14, pp.49-54, Jul 2014*

[4] G. Hachez, "A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards," PhD Thesis, Université Catholique de Louvain, 2003

[5] J. Rajendran, Y. Pino, O. Sinanoglu, and R. Karri, "Security analysis of logic obfuscation" in Proc. Des. Autom. Conf. DATE, pp. 83–89, 2012

[6] R.S Chakraborty, S. Bhunia, "HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.28, no.10, pp.1493,1502, Oct. 2009

[7] R. Torrance, D. James, "The state-of-the-art in semiconductor reverse engineering," *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE* , pp.333,338, 5-9 June 2011

[8] M. Brzozowski, V.N. Yarmolik, "Obfuscation as Intellectual Rights Protection in VHDL Language," *Computer Information Systems and Industrial Management Applications, 2007. CISIM '07. 6th International Conference on* , vol., no., pp.337,340, 28-30 June 2007

[9] U. Meyer-Baese, E. Castillo, G. Botella, L. Parrilla, and A. Garcia, "Intellectual Property Protection (IPP) using Obfuscation in C, VHDL and Verilog Coding, "*In proceedings of Independent Component Analyses, Wavelets, neural Networks, Biosystems and Nanoengineering IX*, SPIE, 2011

[10] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri, "Security Analysis of Integrated Circuit Camouflaging, " *in proceedings of ACM Conference on Computer and Communications Security*, ACM, 2013

[11] SypherMedia, "Syphermedia library circuit camouflage technology," http://www.smi.tv/syphermedia_library_circuit_camouflage_technology.html

[12] J. A. Roy, F. Koushanfar, and I. Markov, “EPIC: Ending piracy of integrated circuits,” in *Design, Automation and Test in Europe*, 2008, pp. 1069–1074.

[13] A. Baumgarten, A. Tyagi and J. Zambreno, "Preventing IC Piracy Using Reconfigurable Barriers" *Design & Test of Computers, IEEE*, vol.27, no.1, pp.66,75, Jan.-Feb. 2010

[14] E. Jung, C. Hung, M. Yang, and S. Choi, "An Locking and Unlocking Primitive Function of FSM-modeled Sequential Systems Based on Extracting Logic Property,” Int. Journal of Information (INFORMATION), Vol. 16, No. 8(B), pp. 6279-6290, August 2013.

[15] M.T. Rahman, D. Forte, Q. Shi; G.K. Contreras, M. Tehranipoor, "CSST: Preventing distribution of unlicensed and rejected ICs by untrusted foundry and assembly," *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014*, pp.46,51, 1-3 Oct. 2014

[16] A. Basak, Y. Zheng, S. Bhunia, "Active defense against counterfeiting attacks through robust antifuse-based on-chip locks," *VLSI Test Symposium (VTS), 2014 IEEE 32nd* , vol., no., pp.1,6, 13-17 April 2014

[17] S. Davidson, “ITC’99 benchmark circuits - preliminary results,” in *IEEE International Test Conference*, Atlantic City, NJ, USA, September 1999, p. 1125.

[18] S. M. Plaza and I. Markov, “Solving the third-shift problem in IC piracy with test-aware logic locking,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[19] S. P. Skorobogatov and R. J. Anderson, “Optical fault induction attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, San Fransisco CA, USA, August 2002.

[20] L. Bossuet and D. Hely, “SALWARE: Salutary hardware to design trusted IC,” in *Workshop on Trustworthy Manufacturing and Utilization of Secure Devices, TRUDEVICE*, 2013.

[21] F. Koushanfar, “Provably Secure Active IC Metering Techniques for Piracy Avoidance and Digital Rights Management”, IEEE Transactions on Information Forensics and Security, vol.7, no.1, pp.51-63.