Reversible Denial-of-Service by Locking Gates Insertion for IP Cores Design Protection

Brice Colombier, Lilian Bossuet
Hubert Curien Laboratory, UMR CNRS 5516, University of Lyon
42000 Saint-Étienne - France
{b.colombier, lilian.bossuet}@univ-st-etienne.fr

David Hély
LCIS, Grenoble Institute of Technology
26000 Valence - France
david.hely@lcis.grenoble-inp.fr

***Abstract*—Nowadays, electronics systems design is a complex process. A design-and-reuse model has been adopted, and the vast majority of designers integrates third party intellectual property (IP) cores in their design in order to reduce time to market. Due to their immaterial form and high market value, IP cores are exposed to threats such as cloning and illegal copying. In order to fight these threats, we propose to achieve functional locking, equivalent to a triggerable and reversible denial-of-service. This is done by inserting locking gates at specific locations in the netlist, allowing to force outputs at a fixed value. We developed a new method based on graph exploration techniques for locking gates insertion. It selects candidate nodes ten thousand times faster than state-of-the-art fault analysis-based logic masking techniques. Methods are then compared on ISCAS'85 combinational benchmarks.**

Keywords—Intellectual property protection, logic masking, functional locking, graph analysis.

I. Introduction

Electronics system design has become a complex and demanding task. Due to a shorter time to market, a design-and-reuse paradigm has been widely adopted. It allows system integrators to design electronics systems faster, using functional building blocks. Companies have specialized in providing these pieces of design, known as intellectual property (IP) cores. The question of a fair IP cores distribution system, however, remains open. Indeed, an IP core designer must disclose the design entirely to the system integrator. Since IP cores can be extremely valuable and are provided as data file, they are a prime target for people with malicious intent. Multiple cases of counterfeiting and non-contracted overbuilding have been reported in recent years, and the trend is growing [1], [2], [3]. Facing such threats, the development of an efficient design data protection scheme is necessary.

An increasingly studied solution is to make the system unusable unless it has been unlocked beforehand. The activation procedure is initiated by the designer. In case the design has been obtained illegally, it remains locked and unusable, until the activation procedure is carried on. To ensure appropriate security, the activation requires a key. This secret key is delivered by the designer, who can thus precisely control how many instances of the design have been instantiated. In order to lock the design, a solution is to insert extra logic gates on specific nodes, that will modify them if the wrong values are applied to the gates' inputs. For instance, XOR/XNOR gates can be used to invert the value of the nodes if the wrong key is applied [4]. In this first article, however, gates are placed randomly. A better placement algorithm, based on fault analysis [5], allows the designer to select more suitable nodes to lock, in order to have a stronger locking power with lower area overhead.

Instead of using XOR or XNOR gates to cause disturbances to the circuit, we propose to achieve a reversible denial-of-service by forcing the outputs to a fixed value if the wrong key is applied. For this purpose, (N)AND and (N)OR gates are used instead, and inserted as deep as possible in the netlist, yet achieving total functional locking. These logic gates are used to force specific nodes to logical 0 or 1. We developed an algorithm, based on graph exploration, to select the best nodes in a netlist for locking gates insertion. Our method enables us to select optimal candidate nodes ten thousand times faster than the fault analysis-based technique. Moreover, since it conducts an exact analysis and is not based on any simulation, the optimal subset of nodes to modify is reached, and the analysis does not rely on any external software.

The remainder of this paper is organized as follows. Section II provides background of active protection schemes based on extra gates insertion. It gives formal definitions for logic masking and functional locking, that are the two protection methods investigated here. Section III presents the new insertion method used to select the nodes of a netlist for functional locking. A comparison is presented in Section IV. It aims at showing the differences between our insertion method and the state-of-the-art. Section V discusses design challenges that must be addressed during protection schemes development. Finally, the paper is concluded in Section VI.

II. Related work

In 2007, the first paper describing an active protection scheme [6] proposes to add extra dummy states before the original start state of the IP core's finite state machine (FSM). A key is required to allow transition through the extra states. Therefore, access to the original FSM was controlled. This way, access to the normal behaviour of the system depends on a secret key, that is only available from the designer. This solution, however, is cryptographically weak, and requires the original design to contain an FSM. Another option is to act on the combinational part of the circuit, and add XOR/XNOR gates at specific locations. Similarly, applying the wrong key on these gates inverts the associated value, and therefore the behaviour of the circuit is altered. This is the principle of logic masking.

Logic masking: XOR or XNOR gates are inserted on the data path in order to change the logical behaviour of the circuit if the wrong key is applied to these additional gates. In this case, some internal nodes have their value inverted. Ideally, the output obtained when a wrong key is applied is not correlated with the original output.

This was first proposed in [4], in which the authors place logic masking gates randomly. Using the random placement method, the correlation between the normal and masked output reduces slowly with respect to the number of masking gates inserted, and the area overhead to obtain low correlation is not negligible. A better placement technique was then necessary. In [5], the authors elaborate on this idea, and propose another method to place the masking gates. The point is to place these gates more efficiently, in order to reduce correlation faster and thus limit the area overhead for an identical level of disturbance. They develop a placement method based on fault analysis, called logic encryption. In order to determine on which nodes of a design XOR or XNOR gates should be placed, they compute a metric, named Fault Impact, for each node. This metric takes into account the number of patterns that detect a s-a-0 and s-a-1 fault for this node, and the total number of output bits that get affected by these faults. Then, the node for which the Fault Impact is maximal is selected. An XOR or XNOR gate is added on it, according to a user defined key, and the process is carried out again until the number of output bits that are affected by a fault reaches 50% on average. This criterion is fulfilled when the Hamming distance between correct and masked output vectors is close to 50%. Once this value is obtained, the logic is considered as ”encrypted”, according to [5] (Section V-C will discuss it in more details). This method can not be referred to as logic encryption though, due to the lack of security proofs and the absence of ciphering and deciphering steps. Instead, this protection technique should be classified as logic masking. What we propose here is to use another method to alter the outputs of the system, called functional locking:

Functional locking: a triggerable and reversible denial-of-service. The circuit can be rendered useless, and then come back to its normal behaviour. This is achieved by forcing the outputs of the netlist to logical 0 or 1 after inserting AND, NAND, OR or NOR locking gates in the netlist.

In both cases, a method is required to select the appropriate nodes to act on, i.e. the ones on which an XOR or XNOR will be placed for masking, or a locking gate for functional locking. In the next section, we present the placement method we developed. It detects sequence of gates in the netlist that can propagate a locking value to one or more outputs.

III. Proposed gates-insertion method for functional locking: Graph exploration for gates sequences

A. Principle

In order to force the output of a logic gate to logical 0 or 1, a specific value must be present on one of its inputs. For instance, the output of a NAND gate is forced to 1 if one input is set to 0. Similarly, the output of an OR gate is forced to 1 if one input is set to 1. Therefore, it follows that all logic gates are able to propagate a locking value if one of their input is set to the right value.

AND: Set one input to **0** forces the output to **0**
NAND: Set one input to **0** forces the output to **1**
OR: Set one input to **1** forces the output to **1**
NOR: Set one input to **1** forces the output to **0**

The AND and NAND gates can propagate a locking value if one of their inputs is set to 0. The OR and NOR gates can propagate a locking value if one of their inputs is set to 1. Hence, it is the value on one of the inputs that determines if a gate propagates the locking value. This value only depends on the preceding gate. An example showing how a sequence of gates (AND → NAND → OR) can propagate a locking value is shown in Fig. 1:

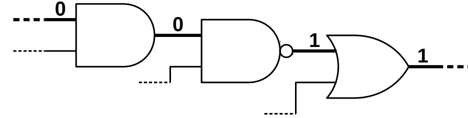

Fig. 1. Propagation of a locking value in a sequence of gates

In order to lock the outputs of a netlist, we must identify such sequences of gates that lead to an output. For that, we compute the two following features for all nodes in the netlist:

- V_{forced}: this is the value at which the node will be forced at. It depends on which type of logic gate precedes the node. For example, if the node is the output of an OR gate, then $V_{forced} = 1$.
- V_{locks}: this is the value at which the node should be forced to propagate the locking It depends on which type of logic gate succeeds to the node. For instance, if the node is the input of a NAND gate, then $V_{locks} = 0$. It should be noted that if a node has a fan-out higher than 1 and spans NAND and OR gates for example, then $V_{locks} = \{0, 1\}$.

A summary of the V_{forced} and V_{locks} values for a node G depending on the type of the preceding and following gates is given in Table I.

Preceding gate	$V_{forced}(G)$
AND	0
NAND	1
OR	1
NOR	0

Following gate	$V_{locks}(G)$
AND	0
NAND	0
OR	1
NOR	1

TABLE I. V_{forced} AND V_{locks} VALUES FOR A NODE G, DEPENDING ON THE PRECEDING AND FOLLOWING LOGIC GATE

It follows that, for sequences that propagate a locking value, Criterion 1 is fulfilled for all the nodes in the sequence.

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

To represent the netlist more easily and identify interesting sequences of gates, we use graph exploration techniques. They are presented in the following subsection.

B. Graph building

First, a directed acyclic graph is generated from the netlist. Logic gates are represented as follows: inputs and outputs are vertices connected by edges labelled after the logic gate type. This is illustrated in Fig. 2.

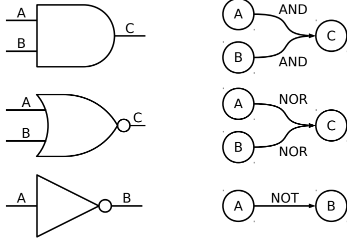

Fig. 2. Logic gates conversion to vertices and edges

This process is carried out on the original netlist. An example of this conversion is shown in Fig. 3.

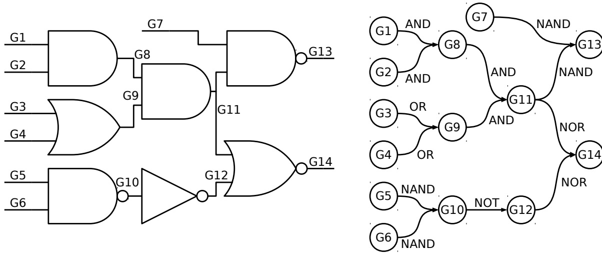

Fig. 3. Sample from a netlist and its equivalent graph

In order to identify interesting sequences in the netlist, we use the previously built graph and compute V_{forced} and V_{locks} for each vertex. Here are the values of V_{forced} and V_{locks} for all the nodes of the netlist in Fig. 3:

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	—
G7	—	0	G14	0	—

TABLE II. EXAMPLES OF V_{forced} AND V_{locks}

According to these values, nodes that do not belong to locking sequences are deleted. Inputs and output nodes are retained. Then nodes that do not fulfil Criterion 1, i.e. $V_{forced} \notin V_{locks}$ are removed from the graph, except if one of their direct successors fulfils Criterion 1 or is an output. Incoming edges should be removed in that case though, because the node cannot propagate the locking value. The outcome is a graph in which all connected vertices can propagate a locking value. This graph, however, is usually disconnected. Connected components that do not include an output of the netlist are removed, since they are not useful for functional locking.

In the previous example, G1 to G6, G13 and G14 are retained since they are inputs or outputs. G9, G10 and G12 do not satisfy Criterion 1, but G9 is retained since its direct successor, G11, fulfils Criterion 11. G9 incoming edges are deleted though. G12 is retained too since its successor, G14, is an output. G3, G4, G5 and G6 are then in a connected component with no outputs, and are therefore deleted. Eventually we obtain the graph shown in Fig. 4. A sequence of gates that can propagate a locking value is also highlighted.

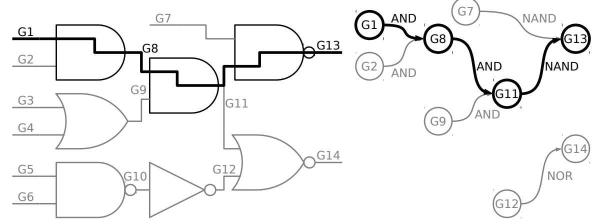

Fig. 4. Sample from a netlist and its equivalent graph after analysis. One locking sequence is highlighted

C. Graph analysis for optimal locking nodes selection

The point is now to select the nodes to lock. This selection process is independent of the Boolean function. It will be taken into account later, when locking gates are inserted (Section III-D). The three different types of connected components that compose the final graph are presented in Fig. 5.

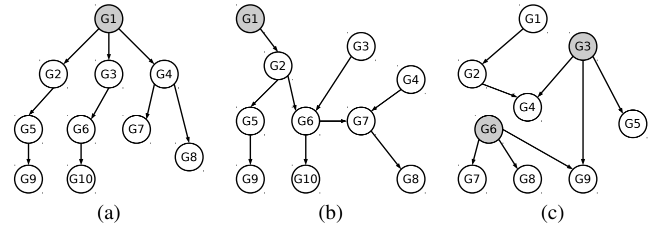

Fig. 5. The three different types of connected components. The optimal vertex to select for functional locking is shown in grey

The case shown in Fig. 5(a), where there is only one source vertex, is rare. A source vertex has an empty in-neighbourhood. Selecting the locking node here is trivial: since the graph is directed and acyclic, only one vertex has all the others as successors. Forcing this node will lock all the outputs of the connected component. In Fig. 5(a), forcing G1 will lock the outputs G7, G8, G9 and G10. The second situation, shown in Fig. 5(b) is more common: multiple source vertices span multiple outputs. Only one of them, however, spans all the outputs of the connected component. Therefore, the corresponding node is selected. If several vertices span all the outputs, then the farthest vertex from all the outputs is the best candidate. It is the case in Fig. 5(b) where G1 and G2 span all the outputs. G1, however, is the farthest vertex from the outputs (G8, G9 and G10) and is preferred to G2. The last case, depicted in Fig. 5(c) requires more computation. Multiple vertices span multiple outputs, but none of them spans them all. The best nodes to select are the ones that span the greatest number of outputs. The number of outputs among each source vertex successors is computed, and source vertices are sorted accordingly. Then they are greedily selected until all the outputs are locked. In the graph shown in Fig. 4, the nodes to lock are G1 and G12. The complete procedure for graph building and analysis is summarized in Algorithm 1.

Algorithm 1: Optimal nodes selection by graph analysis

Input: Netlist *file*
Output: List of nodes to lock *list_lock*

```
// Build graph
for line in file do
    Add vertex to graph for every new node in line
    Add edges to graph for Boolean function in line
    if "OUTPUT" in line then
        Add vertex to list_outputs

for vertex in graph do
    Compute V_forced(vertex) and V_locks(vertex)

// Select vertices
for vertex in graph do
    delete ← False
    if vertex is neither an input nor an output then
        if V_forced(vertex) ∉ V_locks(vertex) then
            delete ← True
            for succ in successors(vertex) do
                if V_forced(succ) ∈ V_locks(succ) then
                    delete ← False
                    Remove vertex incoming edges
    if delete = True then
        Remove vertex from graph

// Identify connected components
list_cc ← Clustering(graph)

// Analyse connected components
for cc in list_cc do
    if cc contains no output then
        Remove cc from graph
    else
        Identify source_vertices in cc
        if #source_vertices = 1 then
            add source to list_lock
        else
            Identify outputs in cc
            for source in source_vertices do
                Compute distance from source to outputs
            if one source spans all outputs then
                add source to list_lock
            else if multiple source spans all outputs then
                add farthest source to list_lock
            else
                while some outputs are unlocked do
                    add the source that spans the greatest
                    ↪ number of outputs to list_lock
Return: list_lock
```

D. Locking gates insertion

We now have a list of nodes to lock, and their respective V_{locks} value. If $V_{locks} = 0$, an AND or a NOR gate is inserted. If $V_{locks} = 1$, an OR or a NAND gate is inserted. In the previous example, we insert a NOR gate on G1 to force it at 0, and an OR gate on G12 to force it at 1. Two locking inputs are added: K1 and K2. The modified netlist is shown in Fig. 6.

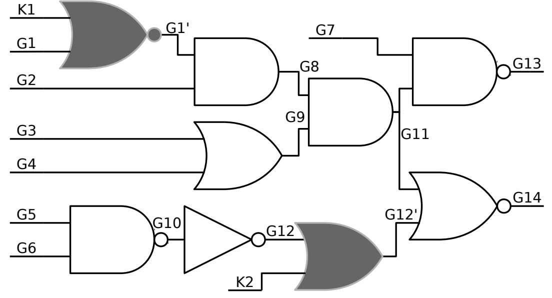

Fig. 6. Modified sample from a netlist with locking gates inserted

In case $V_{locks} = \{0, 1\}$, then the node has a fan-out superior to 1, and two locking gates must be inserted. One AND or NOR gate is inserted to lock part of the fan-out to 0. One NAND or OR gate is inserted to lock part of the fan-out to 1.

IV. Experimental results

A. Real case

The implementation of Algorithm 1 has been done in Python. The *igraph* extension is used for graph analysis. We validated our method on ISCAS'85 benchmarks, and typically obtained the type of connected component shown in Fig. 7. Forcing G27 (light grey vertex) to 0 locks the seven outputs (white vertices) contained in this connected component: G809, G656, G820, G636, G845, G704 and G717. For instance, forcing G27 to 0 will force G3176 to 0, that will force G635 to 0, that will force G636 to 1.

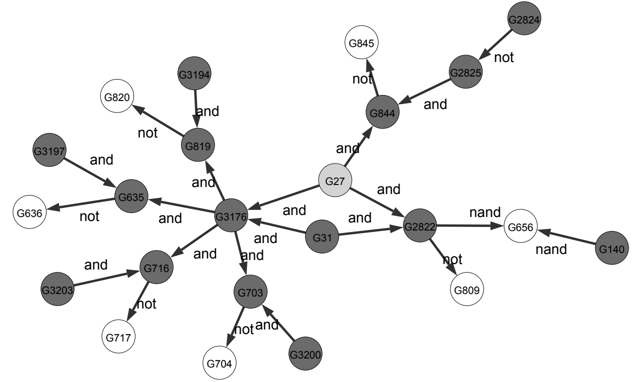

Fig. 7. A connected component from the graph obtained after c5315 netlist analysis

B. Security analysis

Correlation is used to evaluate the security of the protection schemes. Let S be the output of the system. $S_{protected}$ is the output when the protection scheme is active. Correlation is computed using Pearson's correlation coefficient:

$$\rho_{S,S_{protected}} = \frac{cov(S, S_{protected})}{\sigma_S \sigma_{S_{protected}}}$$

When considering functional locking though, since the outputs are stuck at a fixed value, $\sigma_{S_{protected}} = 0$. Thus Pearson's correlation coefficient cannot be computed. Nevertheless, since

the output is fixed, it provides no information about the underlying locking scheme. This is why functional locking can be analysed as if the correlation had a zero value.

	Key size	Random [4]	Fault analysis [5]	Graph analysis
c432	32 bits	0.272	0.012	0
7 outputs	64 bits	0.153	0.019	0
189 nodes	128 bits	0.026	0.014	0
c5315	32 bits	0.902	0.554	0
123 outputs	64 bits	0.873	0.357	0
2362 nodes	128 bits	0.820	0.277	0
c7552	32 bits	0.952	0.254	0
108 outputs	64 bits	0.920	0.235	0
3612 nodes	128 bits	0.761	0.217	0

TABLE III. CORRELATION OBTAINED FOR THREE BENCHMARKS AND DIFFERENT PLACEMENT TECHNIQUES WITH RESPECT TO THE KEY SIZE

As expected, increasing the key size reduces correlation and makes the protection more effective (see Table III). The main drawback that is revealed here is that the efficiency of logic masking drops drastically for large circuits. For instance, using a 128-bit key instead of a 32-bit one only reduces correlation from 0.254 to 0.217 using fault analysis on c7552, that comprises 3612 nodes. It shows that for large netlists, correlation cannot reach optimal values, at least not with keys of reasonable length. It follows that the security can not be proven for such circuits. On the contrary, by identifying gates sequences in the netlist and applying functional locking, it is proven that outputs are locked by forcing the candidate nodes to a specific value. Consequently, since security requirements are so hard to fulfil with logic masking, one should use functional locking instead, and rely on a separate cryptographic function to establish provable security, as it has been proposed in [5].

C. Area overhead

The second criterion used to evaluate the protection schemes is the area overhead, since we want it to be as low as possible. For a fair comparison, ISCAS'85 combinational benchmarks have been used as references. We give the amount of extra gates. Results are depicted in Fig. 8. Graph exploration-based functional locking requires slightly more than 3% extra gates on average to achieve total functional locking of the netlist. For small circuits such as c432, comprising 189 nodes, functional locking requires 3.27% extra gates, when logic masking requires three times more gates : +10,46%. For large circuits, the overheads come closer and are similar for logic masking (+5% for c5315, +1.62% for c7552) or functional locking (+3.57% for c5315, +2.70% for c7552). Unlike logic masking, however, that is only partially efficient for large circuits, functional locking ensures total locking.

D. Computation time

Computation times are given here for the placement method described in [5] and for graph analysis. The workstation we used embeds an Intel Core i5-4570 operating at 3.20GHz and a 16Gb RAM. A log plot of the computation times for all the benchmarks is given in Fig. 9. As we can see, the proposed graph exploration method is ten thousand times faster than fault analysis on average. This is a valuable result. Indeed, those protection schemes could then be integrated EDA tools, in order

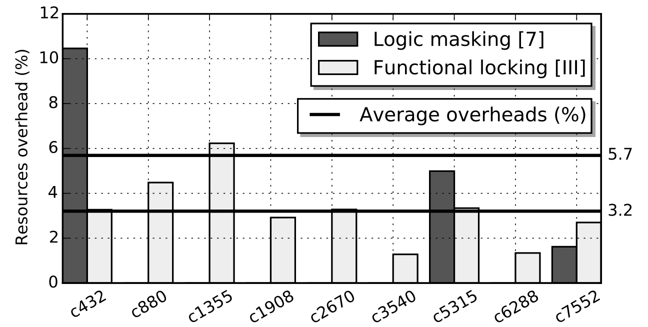

Fig. 8. Area overhead for the two protection schemes

for the designer to be able to add protection to its design on the fly. Even for the bigger netlist, c7552, that contains more than 3600 nodes, the graph exploration method takes only 2.5s to identify gates sequences and return a list of candidate nodes for functional locking. Conversely, fault analysis takes several hours on the same netlist.

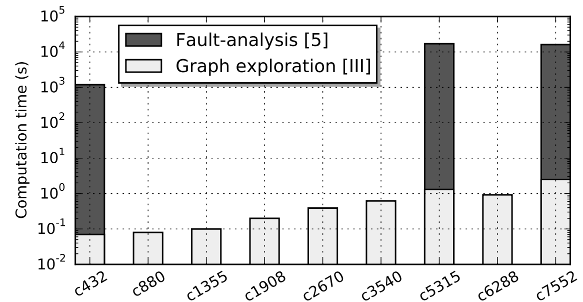

Fig. 9. Computation time required to analyse the original netlist for the two techniques

The main advantage of the proposed methods is to not depend on any external software to conduct the analysis. Only Python scripts were used to analyse the netlists, providing optimal locking gates placement and generating the modified, lockable netlist. It considerably speeds up the analysis in comparison with fault analysis-based method, that employs fault analysis and simulation tools.

V. DISCUSSION

A. Exact or simulation-based placement methods

The placement technique proposed in [5] selects the best candidates with the help of a fault simulator. It evaluates the Fault Impact after applying one thousand random input patterns to the netlists. It requires circuit simulation, however, to derive the nodes to modify for logic masking. Therefore, this method is sub-optimal, since it is based on a partial exploration of the design space. In comparison, our technique based on graph exploration for gates sequences is optimal. Consequently it always provides the optimal subset of nodes that should be locked.

B. Security margin for functional locking

In III, we only gave details about how to select as few nodes as possible in order to lock all the outputs of a design. In the

context of real-life designs protection, however, the designer may want to add extra locking gates on the locking propagation path. This ensures that the locking value is set by multiple gates, and is harder to cancel. Actually, all the vertices that are present in the final subgraphs can be modified to lock outputs. Therefore, the designer can choose precisely the number of gates to be involved in the locking process. For example, as shown in Fig. 7, G27 can lock all the outputs. All the other nodes (dark grey vertices) are candidates too, although they do not lock as many outputs as G27. They can be used to add redundancy to the locking scheme, and provide better security through stronger locking. As it is the case for all the proposed protection schemes, the final implantation is a trade-off between security and area, power and timing overhead. The minimum and maximum number of locking gates that can be selected for each netlist is shown in Fig. 10.

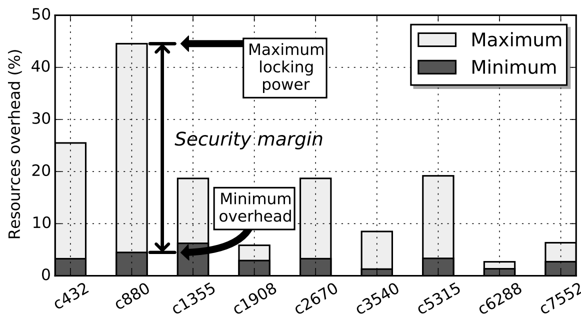

Fig. 10. Resources used to protect the netlist by functional locking using the graph analysis method

C. On the necessary separation between the security primitive and the locking/masking module

The design flow of most of the protection schemes found in the literature is the following: after designing a strong and efficient locking scheme, the authors do their best to highlight cryptographic properties of their system. For instance in [5], the authors look for a 50% Hamming distance between the original and masked output vectors. There are some circuits, however, for which this criterion can be satisfied, that bring absolutely no security, cryptographically speaking. Therefore, we believe that instead of trying to develop cryptographic and locking functions out of the same module, those two essential parts of the protection scheme must be explicitly separated. This separation is only suggested in [5]. On the one hand, a provably secure cryptographic primitive should be selected to guarantee safe access to the protection scheme. Examples of lightweight ciphers can be easily found, and some of them have been extensively studied, such as PRESENT [7]. Moreover, the advantage of using a dedicated cipher is that the key can be different for all instances of a given design, and can be derived from a PUF for example [8]. This is an important security requirement for design data protection, since unlocking one device should not be of any help in unlocking others. On the other hand, a locking scheme with a small footprint is used. Its only purpose is to make the system useless when the wrong key is applied to the cryptographic function, nothing more. This configuration allows the designer to choose the security and locking functions independently. A crucial point here is how these two primitives will be combined to form the complete protection scheme. A one-bit signal triggering the activation of the locking scheme is not suitable. It could compromise the security of the whole scheme, since bit flipping is achievable using a laser for example [9]. A multiple-bit signal should be used instead, to avoid this issue. The description of a complete protection scheme, however, is beyond the scope of this paper.

VI. Conclusion

This paper discusses a new approach to select candidate nodes of a netlist for functional locking. The proposed method has proven to be much more computationally effective than the state-of-the-art, yet achieving similar area overhead. In addition, it conducts an exact analysis using clearly defined parameters, instead of using incomplete simulation. Next, the necessity of a strict distinction between security and locking functions was highlighted, and should be taken into account in future protection schemes design. Thus paired with a secure yet lightweight cryptographic function, functional locking techniques form a strong design data protection scheme, that allows the designer to keep control on its design after it has been sold.

Acknowledgement

The work presented in this paper was realized 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”.

The authors would also like to thank Jeyavijayan Rajendran [5] for providing his results and netlists for comparison.

References

[1] Frontier-Economics, “Estimating the global economic and social impacts of counterfeiting and piracy,” Business Action to Stop Counterfeiting and Piracy (BASCAP), Tech. Rep., 2011.

[2] IHS Technology. (2012, April) Top 5 most counterfeited parts represent a $169 billion potential challenge for global semiconductor market. IHS Technology.

[3] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, “Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain,” *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[4] 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.

[5] J. Rajendran, H. Zhang, G. S. Rose, Y. Pino, O. Sinanoglu, and R. Karri, “Fault analysis-based logic encryption,” *IEEE Transactions on Computers*, October 2013.

[6] Y. Alkabani and F. Koushanfar, “Active hardware metering for intellectual property protection and security,” in *USENIX Security*, Boston MA, USA, August 2007, pp. 291–306.

[7] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, “PRESENT: an ultra-lightweight block cipher,” in *International Workshop on Cryptographic Hardware and Embedded Systems*, Vienna, Austria, September 2007, pp. 450–466.

[8] L. Bossuet, X. Ngo, Z. Cherif, and V. Fischer, “A PUF based on transient effect ring oscillator and insensitive to locking phenomenon,” *IEEE Transaction on Emerging Topics in Computing*, vol. 2, no. 1, pp. 30–36, 2013.

[9] 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.