Closing Leaks: Routing Against Crosstalk Side-Channel Attacks

Zeinab Seifoori
seifoori@ce.sharif.edu
Sharif University of Technology
Tehran, Iran

Seyedeh Sharareh Mirzargar
seyedeh.mirzargar@epfl.ch
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland

Mirjana Stojilović
mirjana.stojilovic@epfl.ch
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland

ABSTRACT

This paper presents an extension to PathFinder FPGA routing algorithm, which enables it to deliver FPGA designs free from risks of crosstalk attacks. Crosstalk side-channel attacks are a real threat in large designs assembled from various IPs, where some IPs are provided by trusted and some by untrusted sources. It suffices that a ring-oscillator based sensor is conveniently routed next to a signal that carries secret information (for instance, a cryptographic key), for this information to possibly get leaked. To address this security concern, we apply several different strategies and evaluate them on benchmark circuits from Verilog-to-Routing tool suite. Our experiments show that, for a quite conservative scenario where 10-20% of all design nets are carrying sensitive information, the crosstalk-attack-aware router ensures that no information leaks at a very small penalty: 1.58–7.69% increase in minimum routing channel width and 0.12–1.18% increase in critical path delay, on average. In comparison, in an AES-128 cryptographic core, less than 5% of nets carry the key or the intermediate state values of interest to an attacker, making it highly likely that the overhead for obtaining a secure design is, in practice, even smaller.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Hardware → Reconfigurable logic and FPGAs**.

ACM Reference Format:
Zeinab Seifoori, Seyedeh Sharareh Mirzargar, and Mirjana Stojilović. 2020. Closing Leaks: Routing Against Crosstalk Side-Channel Attacks. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373087.3375319

1 INTRODUCTION

Parallel nature of field-programmable gate arrays (FPGAs) enables them to offer superior processing power than modern processors, and makes FPGAs perfect hardware-acceleration platforms for a large spectra of applications. With increased application space and technology scaling, FPGAs continue to grow in size and number of available resources. Consequently, FPGAs today can accommodate extremely large heterogeneous designs. In large development projects, modularisation and design reuse are the key to meeting delivery deadlines while ensuring the validity of the entire design. Often, in the absence of in-house design modules (cores) for some of the required design functionalities, companies opt for purchasing IP cores from sources specialized in IP-core development or for outsourcing a part of their development to other companies. As a result, large designs are often assembled from modules originating from various sources, presumably all trusted and performing only the desired functionality. In the absence of means or time to thoroughly test whether all the design modules are free of any malicious code, designers would greatly benefit from this task to be handled by the EDA tools.

Crosstalk coupling between neighboring routing wires is known to affect signal delays [12]: a long wire carrying logical 1 reduces the propagation delay of the signal carried by the adjacent long wire [3]. This phenomenon can not only be observed, but also exploited for side-channel attacks [3]. All that it takes is for an FPGA Trojan to enable a ring oscillator, which has to be conveniently routed so that one of the wires between two ring-oscillator stages is neighboring the victim signal. To validate if the threat of a crosstalk attacks is real, Provelengios et al. examined long wire coupling on various types of wires across three FPGAs in technology nodes from 60 to 20 nm (Cyclone IV, Stratix V, Arria 10); their findings were affirmative [8].

Researchers have been focusing on the feasibility of crosstalk attacks in multi-user setting, such as FPGAs in the cloud, not entirely realizing that higher risk lies in designs assembled from IP cores, acquired in the RTL form and assembled as parts of large and complex designs. One of such IP cores, for example, a USB transmitter/receiver, may contain a hidden FPGA Trojan, actively attempting to pick up signals from neighboring wires. If one of those wires, for instance, carries a secure cryptographic key—required by an encryption core that happens to be an integral part of the design—a crosstalk side-channel attack becomes reality.

One approach to preventing crosstalk attacks could be by identifying all combinational loops, ring oscillators being among them. However, researchers have shown that even without a combinational loop it is possible to synthesize very efficient sensors for measuring crosstalk leakage [2]. Hence, we take a completely different approach. In this paper, we show how PathFinder [7], well-known routing algorithm for FPGAs, can be extended (at minimal cost) to prevent crosstalk FPGA Trojans from performing an attack. With our algorithm, all designers need to do is label nets that carry sensitive information, for instance the nets carrying the cryptographic key or intermediate encryption/hash register states. The router then ensures that all labeled nets are never routed close to any of the signals that originate from potentially untrusted sources, thus effectively preventing the crosstalk attack.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375319

The main contributions of this paper can be summarized as follows:

- To the best of our knowledge, this is the first work that leverages FPGA routing to prevent crosstalk side-channel attacks on FPGAs.
- Four enhancements (or modes of operation) of the PathFinder routing algorithm are presented, some of them more and some less constraining.
- Using benchmarks from Verilog-to-Routing (VTR) suite [9], we experimentally evaluate how routing in secure mode performs, in terms of minimal channel width and critical path delay, compared to the baseline VPR router [9]. The results show that, for a conservative scenario where 10% or 20% of all benchmark nets carry sensitive information, our router ensures that no information can leak at an acceptably small penalty of 1.58–7.69% increase in minimum routing channel width and 0.12–1.18% increase in critical path delay, on average. In comparison, in an AES-128 cryptographic circuit, less than 5% of nets carry the key or the intermediate state values of interest to an attacker [1]. Hence, in practice, these overheads may be even smaller.

In the remainder of this paper, we first address the related work (Section 2). Then, we lay down the background in crosstalk side-channel leakage and PathFinder FPGA routing algorithm (Section 3). In Section 4, we describe our enhancements of PathFinder. Section 5 explains the experimental setup and presents the results, while Section 6 concludes the paper.

2 RELATED WORK

There has been quite a bit of work on VLSI routers that understand crosstalk [10, 11], but surprisingly very little in routers for FPGAs [12]. Wilton, in his crosstalk-aware router for FPGAs, modifies the cost function used by the PathFinder algorithm to include the delay penalty caused by the crosstalk effect [12]. As a result, the router tends to route nets with high criticality away from other nets, thereby lowering the crosstalk effect on the routing delays. Our work is different, because we are not concerned by crosstalk effects across the entire FPGA design. Applying a crosstalk-related cost penalty to all nets and trying to reduce crosstalk everywhere on the FPGA is not our target goal. Moreover, our design constraints are tighter: to avoid crosstalk side-channel leakage, we *must* guarantee that no sensitive net is routed next to an untrusted net [2, 3]. Previously published crosstalk-aware routers [12] do not address security issues, whereas we do.

Huffmire et al. suggest using a spatial isolation mechanism called a moat and a controlled core-to-core communication mechanism called a drawbridge [4, 5], as methods for ensuring separation on reconfigurable devices. To construct moats, they partition the design and place cores in nonoverlapping regions of the chip. The unused space between cores becomes the moat. Inside moats, routing is disabled, except for the signals that use drawbridges to cross moats. The authors report an overhead of 1,000 configurable logic blocks, for the moat of size six and a design of seven cores [4], as well as the maximum decrease in design clock frequency of ~2%. More recently, Yazdanshenas and Betz suggest wrapping the FPGA user applications (also known as roles) with soft shells [13], in which

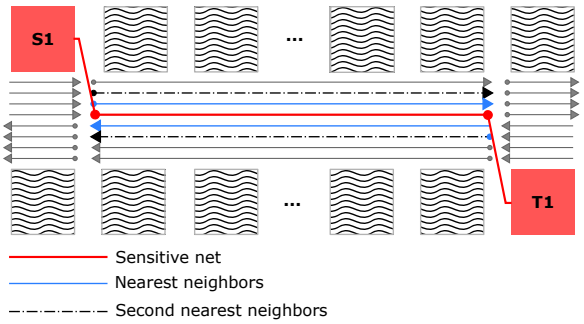

Figure 1: Nearest neighbor wires (in blue) and second nearest neighbor wires (dashed) of a sensitive net, whose routing tree contains a long wire (in red). For simplicity, only the path from the net source node (S1) to one of the sink nodes (T1) is shown. Additionally, neither the short wires nor the vertical routing channels are drawn.

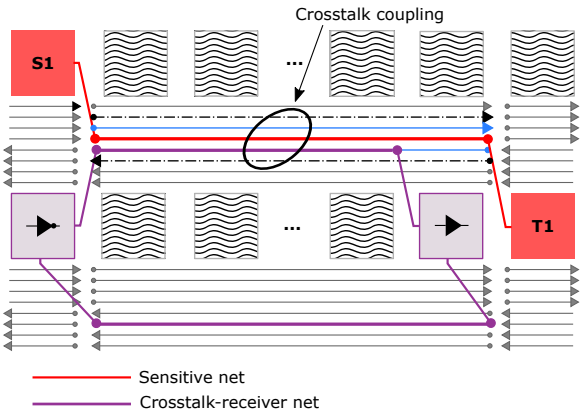

Figure 2: Threat scenario: the ring oscillator of the crosstalk side-channel receiver (in purple) is routed next to a sensitive net (in red). For simplicity, only the path from the net source node (S1) to one of the sink nodes (T1) is shown. Additionally, neither the short wires nor the vertical routing channels are drawn.

the data that has to leave the role is encrypted, ensuring off-role confidentiality. However, this causes about 80% higher latency of the secure bandwidth compared to the regular (unencrypted) traffic. Moreover, the additional resources to implement the encryption reduce the area available for the FPGA roles by about 20%. In our work, no physical separation between the cores is required, while the decrease in critical path delay, as we will show later, remains below 2%, on average.

3 BACKGROUND

3.1 Crosstalk Side-Channel Leakage

Giechaskiel et al. observed that a long routing wire carrying a logical 1 reduces the propagation delay of another adjacent, but unconnected, long wire in the FPGA routing network [3]. The change in the wire delay can be relatively simply measured using a

ring oscillator (sequence of an odd number of inverters closed in a loop) and a frequency counter, thus allowing malicious users to build an FPGA Trojan capable of performing crosstalk side-channel attacks.

In the same work, the authors experimentally demonstrated that the longer the overlap between the neighboring long wires, the more pronounced the crosstalk effect is. This phenomenon is still observable, although 20× weaker, when the transmitter and the receiver wires are separated by a single unoccupied long wire. When the transmitter and the receiver pair are separated even farther, the coupling is too weak and the transmitted data cannot be reliably inferred. Given that it is not the switching frequency of the transmitted signal but its value that determines the wire delay, even constant signals (such as the outputs of registers keeping encryption key) may leak information. Therefore, preventing crosstalk attacks is equivalent to not permitting any malicious net to come close to nets carrying sensitive information.

Fig. 1 illustrates what the nearest and the second nearest neighbors of a wire are: the wires at a distance one or two, respectively. Fig. 2 shows the crosstalk-attack threat scenario, in which the ring-oscillator of the side-channel receiver is routed next to a sensitive net. For simplicity, only the path from the net source node to one of the sink nodes is shown. Additionally, neither the short wires nor the vertical routing channels are drawn.

3.2 PathFinder Routing Algorithm

Although the exact FPGA architecture varies from vendor to vendor, all FPGAs share the same basic structure; they are two-dimensional arrays of configurable logic blocks and hardened computational units separated by unidirectional routing wires. To achieve the best design speed, modern FPGAs typically have a mix of short and long wires, in both horizontal and vertical directions.

The FPGA routing resources are represented as a directed *Routing Resource Graph (RRG)* $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ is a wire or a pin of an FPGA LUT, register, or hardened unit. Each edge $e_{ij} \in E$ is a configurable switch allowing to connect a pin to a wire, or a wire to a wire. Signals to route through G form the design netlist, where every net is defined by its source vertex s_i and the sink vertices $\{t_1, t_2, ..., t_m\}$. A net N_i is routed once the paths from its source s_i to each of its sinks are found; those paths constitute a *routing tree* $RT(N_i) \subset G$ of net N_i. Routing of the entire FPGA design (netlist) is successful if the routing trees of all nets are disjoint in G.

PathFinder is the most common academic and commercial FPGA routing algorithm [7]. It is a negotiated-congestion router, that iterates over all nets, applies A* search to find the shortest routing trees, and incrementally increases the cost of vertices in G so that, eventually, the congestion among the resources is resolved and all the routing trees are disjoint.

4 SECURE ROUTING

Let us introduce the following design module properties: **trusted** and **untrusted**. All nets originating from a trusted design unit (designed in-house or obtained from a trusted source) are considered trusted, whilst all nets originating from potentially untrusted IPs are considered untrusted.

Algorithm 1: Function to route the entire design netlist.

```
Input: Routing resource graph: G = {V, E}
Input: Nets N = {N_i}, i = 1..|N|
Output: Routing trees RT = {RT_i} ⊂ G, i = 1..|N|
while congestionExists and NumIterations < MAX do
    foreach net N_i ∈ N do
        ripUpRoutingTree(RT_i)
        updatePresentCost(RT_i)
        RT_i ← s_i                    // Source node of net N_i
        foreach sink t_{i,j} ∈ N_i do
            P_j ← mazeExpand(RT_i, t_{i,j})
            RT_i ← RT_i ∪ P_j
        updatePresentCost(RT_i)
        setGuards(RT_i)
        blockOccupiedResources(RT_i)
    foreach net N_i ∈ N do
        updateHistoryCost(RT_i)
        resetGuards(RT_i)
```

For every net, one can define a property called *key*, initialized with the identifier of the design module from which this net originates. This enables telling whether two nets originate from the same or two different design units.

One prevention mechanism is to limit the routing of the nets internal to IPs to physically separated regions. However, IPs communicate with the rest of the design, hence some nets must leave the regions where their circuits are placed; moreover, some of those nets may be carrying sensitive information. To distinguish the nets carrying sensitive information from others, we introduce new net property: **sensitive**. For instance, all signals carrying secure encryption key are sensitive. In the most general case, both trusted and untrusted design modules can have sensitive nets.

4.1 Routing Strategies

Our FPGA router is based on the VPR Pathfinder algorithm [9], which we modify to stop routing when both the congestion is removed and a specific correctness criterion is satisfied. Correctness criteria can be more or less strict while achieving the same goal, which is why we propose the following strategies:

(1) **Block-2NN**: the nearest and the second nearest neighbors of the sensitive nets are unoccupied.
(2) **Block-NN**: the nearest neighbors of the sensitive nets are unoccupied.
(3) **Block-Untrusted**: the nearest neighbors of the sensitive nets are never occupied by an untrusted net originating from another design module.
(4) **Lock-NN**: the nearest neighbors of the sensitive nets can only be occupied by the nets having the same key, i.e., the nets originating from the same design module as the corresponding sensitive nets.

Our enhanced FPGA router is designed to work with any of the above strategies.

Algorithm 2: Function `setGuards` that assigns locked/blocked constraints on guard wires.

```
Input: Routing resource graph: G = {V, E}
Input: Net N_i and its routing tree RT_i ⊂ G
Input: Safety level L
if N_i.sensitive = True then
    foreach node v ∈ RT_i do
        if v is a long wire then
            switch L do
                case Block-2NN do
                    foreach w ∈ extendedNeighborhood(v)
                     do
                        w.blocked ← True
                case Block-NN do
                    foreach w ∈ neighborhood(v) do
                        w.blocked ← True
                case Lock-NN do
                    foreach w ∈ neighborhood(v) do
                        if w.locked = True then
                            if w.key ≠ N_i.key then
                                w.blocked ← True
                        else
                            w.locked ← True
                            w.key ← N_i.key
```

Algorithm 3: Function `blockOccupiedResources` that blocks all wires in use by a sensitive net.

```
Input: Routing resource graph: G = {V, E}
Input: Net N_i and its routing tree RT_i ⊂ G
Output: Updated G
foreach node v ∈ RT_i do
    if N_i.sensitive = True then
        v.blocked ← True
    else
        v.blocked ← False
```

4.2 Routing Algorithm

VPR sorts the nets in decreasing order of criticality, which is the amount of the available timing slack on the nets. We keep the same approach, except that we give priority to the sensitive nets, because they impose constraints on the routing of all the other nets. Consequently, we sort the nets so that on top of the list are the sensitive nets, followed by all the other nets. Both the sensitive and the other nets are sorted in decreasing order of criticality. As a result, a noncritical sensitive net will always be routed before a critical nonsensitive net, possibly affecting the design critical path.

In the VPR routing resource graph, every node description contains the list of the node successors (adjacent nodes), for faster graph traversal while searching for the paths. We modify the RRG representation of the long wires by adding two additional lists:

Algorithm 4: Function `resetGuards` that clears locked and blocked flags.

```
Input: Routing resource graph: G = {V, E}
Input: Net N_i and its routing tree RT_i ⊂ G
Input: Safety level L
Output: Updated G
if N_i.sensitive = True then
    foreach node v ∈ RT_i do
        v.blocked ← False
        switch L do
            case Block-2NN do
                foreach w ∈ extendedNeighborhood(v) do
                    w.locked ← False
                    w.blocked ← False
            otherwise do
                foreach w ∈ neighborhood(v) do
                    w.locked ← False
                    w.blocked ← False
```

Algorithm 5: Function `mazeExpand` that finds a path from the net routing tree to one of the net sinks.

```
Input: Routing resource graph: G = {V, E}
Input: Net N_i and its routing tree RT_i ⊂ G
Input: One of the net sink nodes t ∈ V
Input: Safety level L
Output: On success, a path from routing tree RT_i to sink t.
        Otherwise, NULL.
foreach node v ∈ RT_i do
    enqueue(PQ, v)
while priorityQueueEmpty(PQ) = False do
    v = dequeue(PQ)
    if v = t then
        return buildPath(v, RT_i)
    foreach node u adjacent to v do
        toEnqueue ← False
        if u is a long wire then
            if u.blocked = True then
                continue
            else
                goodNeighborhood ← checkNeighborhood(u)
                if goodNeighborhood then
                    if u.locked = True then
                        if u.key = N_i.key then
                            toEnqueue ← True
                    else
                        toEnqueue ← True
        else
            if u.blocked = False then
                toEnqueue ← True
        if toEnqueue = True then
            enqueue(PQ, u)
return NULL
```

Algorithm 6: Function checkNeighborhood that tests if it is safe to consider adding an RRG node to the priority queue.

```
Input: Routing resource graph: G = {V, E}
Input: A node v ∈ V
Input: Safety level L
goodNeighborhood ← True
if L = Block-Untrusted then
    if N_i.trusted = False then
        foreach node w ∈ neighborhood(v) do
            if w.occupancy = 1 then
                k ← whichNetOccupiesNode(w)
                if N_k.sensitive = True then
                    if N_k.key ≠ N_i.key then
                        goodNeighborhood ← False
    else
        if N_i.sensitive = True then
            foreach node w ∈ neighborhood(v) do
                if w.occupancy = 1 then
                    k ← whichNetOccupiesNode(w)
                    if N_k.key ≠ N_i.key then
                        if N_k.trusted = False then
                            goodNeighborhood ← False
```

the nearest neighbors and **the second nearest neighbors**, containing the vertices corresponding to the neighboring wires, as illustrated in Fig. 1. Additionally, we add a property called **locked** and a property called **key** to the long wires. Finally, we add a property **blocked** to the wires in general, to signal that no net should be occupying them if that property is set.

As shown in Algorithm 1, in each routing iteration, the previous routing tree of each net is ripped-up and then re-routed. Afterwards, the cost of the routing resources is updated. To ensure that the neighbors of the routing resources used by the sensitive nets satisfy the routing strategy, *setGuards* (Algorithm 2) and *blockOccupiedResources* (Algorithm 3) functions are called. The latter function blocks all routing resources used by the sensitive nets (except the source and the sink nodes), to prevent them from being used by other nets. At the end of each routing iteration, *resetGuards* function (Algorithm 4) is called to invalidate the *locked* and the *blocked* flags of all routing resources.

Based on the routing correctness criterion, *setGuards* function assigns guarding properties to wires. For **Block-NN** or **Block-2NN**, it blocks the nearest or both the nearest and the second nearest neighbors (`extendedNeighborhood`), respectively. If the routing criteria is **Lock-NN** and the neighboring wire is not locked, the function locks it and assigns to it the key of the sensitive net. However, if the neighboring wire is already locked using another key (because it already has a sensitive net as its neighbor), we block it, to prevent it from being occupied by other nets.

To route a net, *mazeExpand* function (Algorithm 5) normally puts all current node's neighbors on the priority queue. However, to satisfy our safety criteria, the status of each routing resource should now be checked before adding that node on the queue. For instance, if the **blocked** property of the routing node is *True*, that node should not be added. If the **blocked** property of the routing node is *False*, *checkNeighborhood* function (Algorithm 6) is called to examine the neighbors of the current node. In case of **Block-Untrusted** approach, it has to check that no untrusted net, originating from a different module, already occupies a neighbor of the node in question.

Table 1: VTR benchmark details.

No.	Name	# Blocks	FFs	Mult	Mem	FPGA Size
B1	blob_merge	6016	735	0	0	29 × 29
B2	boundtop	2921	1671	0	1	21 × 21
B3	ch_intrinsics	413	233	0	1	8 × 8
B4	diffeq1	434	193	5	0	12 × 12
B5	diffeq2	277	96	5	0	12 × 12
B6	mkDelayWorker32B	5580	2491	0	9	48 × 48
B7	mkPktMerge	226	36	0	3	26 × 26
B8	mkSMAdapter4B	1977	983	0	3	18 × 18
B9	or1200	2963	691	1	2	25 × 25
B10	raygentop	2134	1423	18	1	17 × 17
B11	sha	2212	911	0	0	18 × 18
B12	stereovision1	10366	11789	152	0	41 × 41
B13	stereovision3	174	102	0	0	6 × 6

5 EXPERIMENTAL EVALUATION

We implement our routing enhancements in VTR 7.0 and test them using the Intel Stratix-IV FPGA architecture and 13 VTR benchmarks [6], listed in Table 1. The horizontal routing channels in Intel Stratix IV FPGA are composed of a combination of wire segments of length four and 20, whereas the vertical channels are composed of wire segments of length four and 12. Given that VTR 7.0 cannot model different horizontal and vertical channel configuration, we use a combination of wires of length four (short wires) and 16 (long wires); in each routing channel, 13% of wires are long and 87% of wires are short.

We first place and route every benchmark, to find the minimum channel width. Then, we identify two groups of nets: those whose bounding box width or length are higher than twice the length of a long wire (**long nets**), and those that do not satisfy the above criteria (**short nets**).

To test **Block-2NN** and **Block-NN** approach, we randomly mark 10% and 20% of all the long and of all the short nets as sensitive, run our router, and measure the minimum channel width and the critical path delay. This is a somewhat conservative assumption as, for instance, a standard AES-128 cryptographic core can have about 10,000 nets, out of which less than 5% carry some secret information [1]. To test **Block-Untrusted** strategy, we randomly mark 50% of all the long and 50% of all the short nets as untrusted (the remaining nets are trusted), and choose 10% or 20% of all trusted (resp. untrusted) nets as sensitive. To test **Lock-NN** approach, we model four IPs by randomly assigning 25% of all the long and 25% all the short nets to every IP. Then, we randomly label 10% or 20% of IP nets as sensitive. Table 2 and 3 summarize the results of all the above experiments, averaged over 10 runs.

The results show that using **Block-2NN** strategy causes an increase in the minimum required channel width by 2.73% (resp.

Table 2: Channel width increase with respect to the baseline VPR router, averaged across 10 experiments.

Benchmarks		Experiment 1: 10%		Experiment 2: 20%		Experiment 3: 10%		Experiment 4: 20%	
No.	**Name**	**Block-2NN**	**Block-NN**	**Block-2NN**	**Block-NN**	**Block-Untrusted**	**Lock-NN**	**Block-Untrusted**	**Lock-NN**
B1	blob_merge	0.79%	0.48%	12.7%	2.38%	0.48%	0.48%	3.65%	3.65%
B2	boundtop	2%	2%	3.09%	2.18%	1.82%	1.82%	2.18%	2.18%
B3	ch_intrinsics	2.44%	2.22%	2.67%	2.89%	1.78%	0.67%	2.89%	2.67%
B4	diffeq1	0.68%	0.23%	1.59%	1.59%	0%	0%	2.27%	1.59%
B5	diffeq2	2.05%	3.59%	10.51%	5.9%	3.08%	2.82%	7.18%	6.41%
B6	mkDelayWorker32B	5.28%	3.06%	6.67%	5.69%	1.81%	1.94%	3.06%	3.47%
B7	mkPktMerge	4.64%	6.07%	29.64%	28.57%	4.29%	5.36%	22.5%	13.57%
B8	mkSMAdapter4B	0.66%	0.33%	2.95%	1.31%	0.16%	0.16%	0.82%	0.98%
B9	or1200	1.59%	2.06%	5.71%	3.97%	1.27%	0.63%	4.6%	2.38%
B10	raygentop	0.61%	0.45%	2.88%	2.27%	0.76%	0.15%	1.06%	1.21%
B11	sha	0.54%	0.18%	1.25%	0.89%	0%	0.18%	0%	0.36%
B12	stereovision1	4.66%	3.7%	7.4%	4.52%	2.33%	1.78%	3.01%	3.42%
B13	stereovision3	9.58%	7.5%	12.92%	12.92%	7.08%	4.58%	18.75%	10.83%
	Average	2.73%	2.45%	7.69%	5.78%	1.91%	1.58%	5.54%	4.06%

Table 3: Critical path delay increase with respect to the baseline VPR router, averaged across 10 experiments.

Benchmarks		Experiment 1: 10%		Experiment 2: 20%		Experiment 3: 10%		Experiment 4: 20%	
No.	**Name**	**Block-2NN**	**Block-NN**	**Block-2NN**	**Block-NN**	**Block-Untrusted**	**Lock-NN**	**Block-Untrusted**	**Lock-NN**
B1	blob_merge	-0.03%	0.35%	0.27%	0.47%	0.43%	0.09%	0.65%	0.28%
B2	boundtop	-5.21%	-3.79%	-3.24%	-5.31%	-5.3%	-4.81%	-3.35%	-4.53%
B3	ch_intrinsics	-1.14%	1.42%	-0.15%	-0.97%	-0.38%	3.71%	0.91%	-0.82%
B4	diffeq1	4.68%	3.56%	6.58%	4.22%	5.87%	4.96%	4.67%	3.66%
B5	diffeq2	7.44%	3.91%	3.23%	5.31%	4.27%	4.78%	6.95%	6.08%
B6	mkDelayWorker32B	-0.49%	-0.1%	-0.2%	0.84%	-0.08%	-0.13%	-0.38%	0.51%
B7	mkPktMerge	0.01%	0.1%	0%	0.12%	0.27%	0%	0.04%	-0.17%
B8	mkSMAdapter4B	0.7%	3%	2.41%	4.89%	2.27%	3.23%	3.84%	3.15%
B9	or1200	0.57%	0.54%	1.19%	3.06%	0.66%	1.46%	1.56%	0.91%
B10	raygentop	-1.92%	-5.34%	-5.21%	-2.67%	-5.07%	-2.72%	-0.37%	-3.24%
B11	sha	0.31%	0.68%	1.06%	0.8%	0.24%	0.02%	2.29%	0.8%
B12	stereovision1	-5.12%	-0.94%	0.19%	0.35%	-1.38%	-1.34%	-1%	-3.07%
B13	stereovision3	1.77%	1.67%	1.08%	4.22%	4.66%	1.88%	-5.05%	2.75%
	Average	0.12%	0.39%	0.56%	1.18%	0.50%	0.86%	0.83%	0.46%

7.69%) for 10% (resp. 20%) of sensitive nets. In case of **Block-NN**, given that only the nearest neighbors of the sensitive long wire are blocked, the corresponding values are reduced to 2.45% (resp. 5.78%). **Block-Untrusted** results in 1.91% (resp. 5.54%) versus 1.58% (resp. 4.06%) when **Lock-NN** is used. Therefore, as expected, the most efficient strategy is **Lock-NN**.

Although the blocking of the routing resources can have a negative impact on the design critical path, the accompanying increase in the channel width sometimes leads to better timing. The results show that the impact that our crosstalk-attack-aware routing has on the critical path delay is almost negligible; on average, the critical path delay increases by a value between 0.12%, for **Block-2NN** and 10% of sensitive nets, and 1.18%, for **Block-NN** and 20% of sensitive nets. In future work, we shall measure the change in the critical path when the channel width is fixed and only the routing algorithm changes; we expect to observe even smaller variations.

6 CONCLUSIONS

In this paper, we leverage CAD algorithms to prevent crosstalk side-channel attacks on FPGAs. Unlike previous work, we do not use moats, drawbridges, nor shells around the IP cores, but modify the FPGA routing algorithm to ensure that all sensitive nets are routed away from untrusted signal sources. Four different enhancements to the state-of-the-art PathFinder algorithm are presented in this paper and experimentally tested on benchmarks from Verilog-to-Routing tool. Results achieved are promising: designs with 10% of sensitive nets can be protected at the cost as low as 1.58% increase in minimal required channel width and 0.86% increase in critical path delay, when **Lock-NN** strategy is used. In large and complex designs, only a very small number of nets carry truly sensitive information. Therefore, it is reasonable to expect similar overheads even in industrial-size projects. We believe that this work will greatly encourage FPGA CAD software providers to adapt similar strategies in their tools and thus assure FPGA users that crosstalk side-channel attacks are prevented by design.

REFERENCES

[1] AIST and Tohoku University. 2019. AES Encryption Core. http://www.aoki.ecei.tohoku.ac.jp/crypto/ Accessed: 2019-12-12.

[2] Ilias Giechaskiel, Kasper Rasmussen, and Jakub Szefer. 2019. Measuring Long Wire Leakage with Ring Oscillators in Cloud FPGAs. In *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications*. Barcelona, Spain, 45–50.

[3] Ilias Giechaskiel, Kasper B. Rasmussen, and Ken Eguro. 2018. Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. In *Proceedings of 13th ACM ASIA Conference on Information, Computer and Communications Security*. Songdo, Incheon, Republic of Korea, 15–27.

[4] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White Ryan Kastner, and Tim Sherwood. 2008. Designing Secure Systems on Reconfigurable Hardware. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 13, 3 (July 2008), 44:1–44:24.

[5] Ted Huffmire, Brett Brotherton, Gang Wang, Timothy Sherwood, Ryan Kastner, Timothy Levin, Thuy Nguyen, and Cynthia Irvine. 2007. Moats and Drawbridges: An Isolation Primitive for Reconfigurable Hardware Based Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. Berkeley, CA, USA, 1–15.

[6] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. 2014. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7, 2 (June 2014), 6:1–6:30.

[7] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *Proc. of the 3th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, 111–17.

[8] George Provelengios, Chethan Ramesh, Shivukumar B. Patil, Ken Eguro, Russel Tessier, and Daniel Holcomb. 2019. Characterization of Long Wire Data Leakage in Deep Submicron FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, USA, 292–297.

[9] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. 2012. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, 77–86.

[10] Hariharan Sankaran and Srinivas Katkoori. 2011. Simultaneous Scheduling, Allocation, Binding, Re-Ordering, and Encoding for Crosstalk Pattern Minimization During High-Level Sythnesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19, 2 (Feb. 2011), 217–226.

[11] Ashok Vittal and Malgorzata Marek-Sadowska. 1997. Crosstalk Reduction for VLSI. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 3 (March 1997), 290–298.

[12] Steven J. E. Wilton. 2001. A Crosstalk-Aware Timing-Driven Router for FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, USA, 21–28.

[13] Sadegh Yazdanshenas and Vaughn Betz. 2019. The Costs of Confidentiality in Virtualized FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 10 (Oct. 2019), 2272–2283.