ObfusX: Routing Obfuscation With Explanatory Analysis of a Machine Learning Attack

Wei Zeng[a], Azadeh Davoodi[a], Rasit Onur Topaloglu[b]

[a]*Department of Electrical and Computer Engineering, University of Wisconsin–Madison, Madison, WI, USA*
[b]*IBM, Hopewell Junction, NY, USA*

Abstract

This is the first work that incorporates recent advancements in "explainability" of machine learning (ML) to build a routing obfuscator called ObfusX. We adopt a recent metric—the SHAP value—which explains to what extent each layout feature can reveal each unknown connection for a recent ML-based split manufacturing attack model. The unique benefits of SHAP-based analysis include the ability to identify the best candidates for obfuscation, together with the dominant layout features which make them vulnerable. As a result, ObfusX can achieve better hit rate (97% lower) while perturbing significantly fewer nets when obfuscating using a via perturbation scheme, compared to prior work. When imposing the same wirelength limit using a wire lifting scheme, ObfusX performs significantly better in performance metrics (e.g., 2.2 times more reduction on average in percentage of netlist recovery).

Keywords: routing obfuscation, split manufacturing, explainable artificial intelligence, machine learning

1. Introduction

Manufacturing outsourcing of Integrated Circuits has become more common than ever before because of the high cost of fabricating high-end chips. As a result, security issues including design piracy and hardware Trojans injection may arise when an untrusted foundry is involved in manufacturing. To alleviate these problems, split manufacturing [1, 2] is proposed as a technique where the untrusted foundry only receives and fabricates a partial layout up to a metal layer denoted by a "split level." However, this may still not prevent an attacker to extract the full design, if the layout is not obfuscated or if the split level is too high, as suggested by [1, 3, 4, 5, 6, 7, 8].

Existing techniques on design obfuscation may be classified as two categories: placement-based and routing-based. Placement-based techniques include pin swapping [1], cell insertion [9], and cell location perturbation [3, 10]. Routing-based techniques include routing blockage insertion [4], routing perturbation [11], and wire lifting [12]. The two techniques may also be combined, as in [13].

The key idea of design obfuscation for split manufacturing is to make an attack model fail to identify correct connections above the split level. As for the attack models for split manufacturing, Rajendran *et al.* first proposed the proximity attack [1]. Wang *et al.* proposed a more advanced network-flow-based proximity attack [3], which employs the network flow model that considers more heuristics for better attack performance. Magaña *et al.* proposed a congestion based attack [4], which redefined proximity measures based on the observation that placement and routing congestions are better indicators in large commercial designs. Most recently, Zeng *et al.* proposed a machine learning (ML) attack model [5], trained with empirically-selected layout features that reflect the hints from routing conventions.

In this paper, we propose a novel way to build an obfuscator for split manufacturing, based on recent advancements in the area of "explainability" of ML. We adopt a recent explanatory metric, namely the **SH**apley **A**dditive ex**P**lanation (SHAP) value [14], to analyze the ML attack model proposed in [5]. (The ML attack model is especially suitable for large commercial designs while other attack models (e.g. [3]) would take prohibitively long attack time.)

The SHAP-based analysis reveals to what extent *each* layout feature contributes to correctly predicting *each* individual unknown connection as seen by an untrusted foundry. We then exploit this information to design a SHAP-guided obfuscator against the ML attack model where only truly vulnerable connections are identified and each is obfuscated by just the necessary amount. This results in minimal perturbation to the layout as measured by increase in wirelength and number of perturbed nets. Our obfuscator (named ObfusX) is routing-based and is performed by utilizing perturbation of vias, and by wire lifting schemes. (Placement-based obfuscation was not found to be as effective by our SHAP-based analysis.)

Overall, our contributions can be summarized as follows.

- This is the first work that shows how explainability in machine learning (ML) can be used for obfuscation; while we focus on routing obfuscation for an ML-based split manufacturing attack, our approach is generalizable to build any obfuscator as long as an ML attack model is available.
- We demonstrate the benefits of ObfusX in identifying and focusing on the most vulnerable candidates and obfuscating each by just the right amount, thereby reducing the obfuscation overhead, while having better performance.
- Our results are compared with two prominent prior works, using not only the ML attack, but also an independent network flow-based attack from a recent work.

2. Preliminaries

To build an obfuscator, we use an explanatory model named SHAP to break a ML-based attack. Here, we review the ML attack model and then give a brief overview of SHAP analysis.

2.1. ML Attack Model for Split Manufacturing

Given a metal layer as the split level, the layout is partitioned into public layers, v-pins (as termed in [5]) and private layers from low to high levels. A *split layer* refers to the topmost metal layer available to the attacker; *public layers* refer to all metal on or below the split layer and via layers in between; *private layers* are all metal layers above the split layer and the via layers in between; *v-pins* are vias connecting public and private layers. The attacker has access to the layout (cells, pins, wires, vias) in public layers and all v-pins. The goal of the split manufacturing attack is to predict the connectivity on private layers based on the available layout on public layers.

Recently, a ML-based attack model was proposed for split manufacturing in [5]. To build the ML model, for each pair of v-pins in a design, first a vector of layout "features" was extracted from the public layers. Using these features, the ML model was built based on Bagging of 10 reduced error pruning trees (REPTrees) in Weka [15]. The ML model mapped each v-pin pair with feature vector $\mathbf{x}$ to a probability $f(\mathbf{x}) \in [0, 1]$, indicating how likely the v-pin pair is a "match" (i.e. actually connected to each other on private layers).

For a pair of v-pins, the following features were extracted in [5]. (We refer the reader to [5] for more details.)

- `diffVpinX`, `diffVpinY`: The x- and y-direction differences in the locations of the two underlying v-pins, respectively.
- `manhattanVpin`: The Manhattan distance of the v-pins.
- `diffPinX/Y`, `manhattanPin`: These are similarly defined but for pins (connected to the v-pin pair) at the placement level.
- `totalWireLength`: Wirelengths of wires connected to the v-pin pair on public layers.
- `totalCellArea`, `diffCellArea`: These are sum (or diff) of the average area of output cells and that of input cells.

2.2. SHAP Tree Explainer for Machine Learning

In this work, we adopt a recently proposed explanatory model named SHAP [14], which explains predictions from a ML model. Let $f(\mathbf{x}_i)$ denote the ML prediction output of the i-th testing sample with feature vector $\mathbf{x}_i$. SHAP decomposes the model output as

$$f(\mathbf{x}_i) = \mathbb{E}[f(\mathbf{x})] + \sum_{j=1}^{M} c_{i,j}, \tag{1}$$

where $\mathbb{E}[f(\mathbf{x})]$ is the expected prediction based on all training data, and $c_{i,j}$ is the contribution of the j-th feature of the i-th testing sample, which can be positive, negative, or zero. Each $c_{i,j}$ indicates to what extent the j-th feature deviates the i-th sample's prediction from the average. The SHAP value is proposed as an excellent candidate to compute the $c_{i,j}$s in (1). SHAP values show how each feature contributes to the model output for each testing sample.

A recent extension [16], referred to as *SHAP tree explainer*, shows that the *exact* evaluation of SHAP values can be done in polynomial time exclusively for tree-based models (which is compatible with the aforementioned ML attack model). The SHAP tree explainer does *not* assume feature independence, as feature interactions are already captured in the underlying trees. In this paper, we will use the SHAP tree explainer to analyze the vulnerability of individual v-pin pairs to the attack and use it to guide the obfuscation.

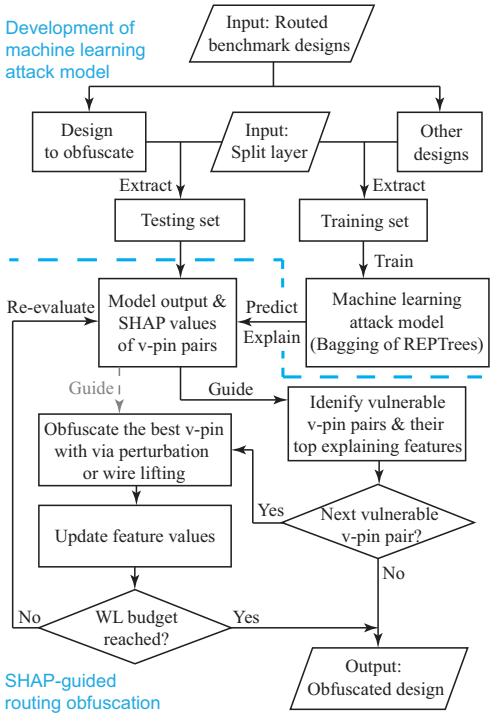


Figure 1: Flow chart of ObfusX.

3. Overview of ObfusX

The core idea of a SHAP-guided obfuscation is to perturb the design, such that a ML attack model would perform worse. As we will show in experiments, such obfuscation also performs well under an independent, non-ML, attack model [3]. This is because both attack models are based on a similar set of design conventions in routing tools that aim to optimize the wirelength, delay, etc. A flow chart of the overall process of ObfusX is shown in Figure 1.

The upper panel shows how the ML model is developed. To generate the training set and testing set for a design to obfuscate (i.e., "target design"), we generate data samples by extracting layout features from routed designs, with the same split layer applied as will be used in manufacturing. All data samples from the target design are allocated in the testing set, which we will

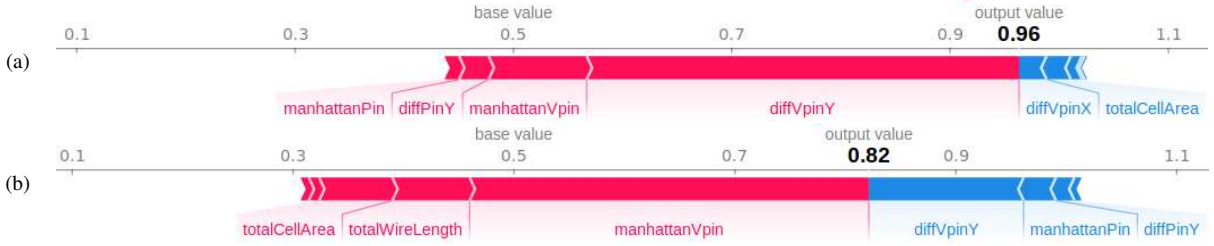


Figure 2: SHAP force plots of two actually-connected v-pin pairs. The pink/blue bars (to the left/right of output values, respectively) quantify to what extend each layout feature positively/negatively contributes to the ML attack that predicts their connectivity. The top contributing features (longest pink bars) may vary from one v-pin pair to another. For example, `diffVpinY` is the most contributing feature in predicting (a) (longest pink bar) while it is actually the most negatively contributing feature to predicting (b) (longest blue bar).

use to monitor the progress and performance of obfuscation. Other designs in the same benchmark suite as the target design are used to generate the training set that will be used to train the attack model. As mentioned in Section 2, ObfusX uses the ML predictor in [5]. With a trained attack model, it predicts how likely each pair of (two) v-pins in the target design could be a match (i.e., are actually connected), which can be interpreted as the vulnerability of the pair to the ML attack.

To develop ObfusX, as shown in the lower panel, the ML prediction for a v-pin pair is fed to the SHAP tree explainer, which generates a set of SHAP values to explain the prediction.

Each SHAP value corresponds to an extracted feature and quantifies to what extent that feature contributes to the ML predictor for that specific v-pin pair. These SHAP values are next analyzed across all actually-connected v-pin pairs to identify the most vulnerable ones to the ML attack, along with the layout features that contribute the most to their individual vulnerabilities.

Next, the output of SHAP analysis guides the actual obfuscation which is done iteratively. ObfusX utilizes two layout perturbation techniques—via perturbation and wire lifting—each of which effectively change the routing and locations of a vulnerable v-pin pair. At each iteration, the most vulnerable v-pin pair is obfuscated if its obfuscation does not violate routing feasibility. Next, the feature vector of the obfuscated pair is updated and consequently its vulnerability is re-evaluated by the attack model (given that the layout has been slightly perturbed). ObfusX then proceeds to obfuscate the next vulnerable pair, until there is no more vulnerable pair, or a budget of wirelength (WL) overhead is reached.

4. SHAP Analysis for One V-pin Pair

Before discussing the details of ObfusX, we first explain how SHAP-based analysis is performed for a single pair of connected v-pins. This helps us to illustrate the true benefits of such analysis in building ObfusX. Consider two connected v-pins from the design `superblue1` with split layer M6. The ML attack model, predicts this pair to be connected with probability 0.96 (which is a relatively high prediction indicating a successful attack if there is no obfuscation).

Figure 2(a) shows the *force plot* generated by SHAP analysis performed on the ML prediction for this pair. The color and length of pink/blue bars show the sign and magnitude of $c_{i,j}$s in Equation (1), respectively. For the pair in Figure 2(a), the analysis breaks down the prediction output of 0.96 as sum of a base value of 0.5 and a total deviation of +0.46. The pink/blue bars correspond to the features which positively/negatively contribute to the model output (i.e., with a positive/negative "force" pushing towards this 0.96 prediction). The length of the bars indicate the degree of contribution such that the sum of the lengths of pink bars (with positive sign) and blue bars (with negative sign) adds up to +0.46.

More specifically, for pair (a), among all its features, `diffVpinY` has the highest SHAP value of around +0.4 (corresponding to the length of its pink bar). Figure 2(b) shows the force plot for a second pair (b). For pair (b), we observe a different feature, i.e., `manhattanVpin` is dominant. Moreover, `diffVpinY`, which was the top feature in (a), has a negative SHAP value in (b), indicating it actually contributes negatively to the prediction of pair (b).

The above example yields the following two key observations to illustrate the unique benefits of SHAP analysis for obfuscation:

1. The vulnerable v-pin pairs can be identified as the ones which have few features with large positive SHAP values.

2. The top feature may vary across individual pairs, implying a different degree or scheme of obfuscation is needed for each.

5. Details of ObfusX

The goal of SHAP-guided obfuscation is to alter the SHAP values such that there will not be any dominant feature with a high positive SHAP. It could mean that obfuscation makes originally dominant features to have a lower positive SHAP value or a negative one.

Our SHAP analysis of design `superblue1` with split layer M6, shows that for about half of the connected v-pin pairs, the SHAP value of `diffVpinY` is consistently dominant (followed by that of `manhattanVpin`). However for the other half of pairs, the distribution of SHAP values over features becomes fuzzy, which suggests that no single feature dominates the model. Such pairs (which do not have any dominant feature) do not need to be obfuscated.

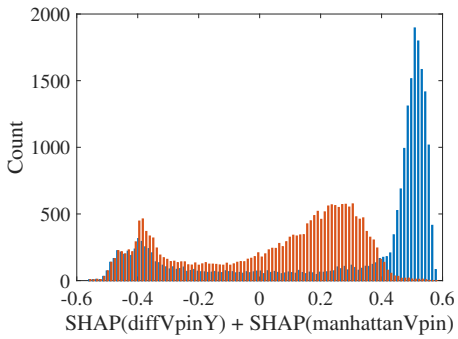


Figure 3: Contributions of top two features, shown as a distribution for all connected v-pin pairs, before (blue) and after (red) obfuscation. ObfusX flattens the distribution and reduces the top contributions.

For the two dominant features (`diffVpinY` and `manhattanVpin`) in the above example, Figure 3 shows the distribution of the combined contribution (i.e., sum of SHAP values) of these two top features, before and after obfuscation. This is when using ObfusX with via perturbation (which will be discussed in detail in Section 5.1). The before-obfuscation distribution is shown in blue and the after-obfuscation one is shown in red. As can be seen, ObfusX flattens the distribution and shifts it to the left (so it decreases the top contributions, making some less positive and some even negative).

Similar to the example of `superblue1` with split layer M6, SHAP-based analysis with the rest of the designs showed that `diffVpinY` and `manhattanVpin` are always the top two contributing features for many of the vulnerable nets when the split layer is even. (For odd split layers `diffVpinY` is replaced with `diffVpinX` because wires are preferred to route vertically on even layers and horizontally on odd layers.) The nets which did not have a dominant feature simply won't need to be obfuscated with SHAP-guided analysis. Therefore, these two features are the only ones which are utilized by ObfusX.

We note, these two dominant features are related to routing which explains our choice to obfuscate the design with routing-based techniques, i.e., via perturbation and wire lifting. However, we note, our general approach is not restricted to routing.

5.1. ObfusX with Via Perturbation

The procedure for via perturbation only considers perturbing v-pin pairs which are determined to be "essential". Essential v-pin pairs are a subset of all connected v-pin pairs, after disregarding trivial cases, e.g., when some v-pins connect to each other using the public layer, hence are easily identifiable by the attacker. ObfusX also ensures feasibility of the routing throughout the process without any area overhead. We first introduce the following which will be used when explaining the algorithm.

5.1.1. Terminology

We introduce the following terminology as shown in Figure 4(a), where the split layer is M4. Wires in all metal layers are shown as horizontal lines and vias as vertical lines.

A *driving pin* is a pin that drives other components in the net. It can be the output pin of a logic cell or that of a primary input.

A *v-pin group* consists of v-pins in the same net that connect to each other using public layers. The v-pins in the same group can be easily identified by an attacker because they are connected in public layers that are available to the attacker.

A *driving v-pin group* is a v-pin group that connects to a driving pin using public layers.

A *non-driving v-pin group* is a v-pin group that does not connect to any driving pin in public layers.

An *essential v-pin pair* (v, v') consists of a pair of v-pins, where v is in a non-driving v-pin group G, and v' is in a driving v-pin group G'. If G' has more than one v-pin, v' is the closest v-pin to v in G'.

5.1.2. Algorithm

We propose an algorithm that perturbs the locations of v-pins based on SHAP values of the top features `manhattanVpin` and `diffVpin`R where R is X for odd split layers and Y for even split layers. This is done iteratively, one v-pin at a time. We first calculate the SHAP values $S(i, j)$ for all essential v-pin pairs i and all features j. Then for each essential v-pin pair i, we take the maximum of the SHAP values over all features j, i.e., $S_{\max}(i) = \max_j S(i, j)$. We only perturb v-pins that satisfy the following criteria:

- The v-pin belongs to an essential v-pin pair $p = (v, v')$, with v and v' in the same net. This is to avoid duplicated or invalid perturbations, e.g. perturbing the same v-pin later when a different v-pin pair is being considered.
- $S_{\max}(p) = S(p,$ `manhattanVpin`$)$ or $S_{\max}(p) = S(p,$ `diffVpin`$R)$. This ensures the essential v-pin pair p is vulnerable, i.e., likely predictable with the top features.
- $S(p,$ `diffVpin`$R) \geq S(p,$ `diffVpin`$R')$, where $R' \in \{X, Y\}$ is the routing direction other than R. This condition ensures the effectiveness of perturbing v or v' in R direction.
- If there are more than one non-driving v-pin group in the net of v and v', then v' in the driving v-pin group is not eligible for perturbation and only v may be perturbed. Otherwise, perturbing v' may affect multiple essential v-pin pairs.

The procedures of SHAP-guided via perturbation are summarized in Algorithm 1. We maintain a list $\mathcal{L}$ of essential v-pin pairs $p = (v, v')$ sorted in decreasing order of $S_{\max}(p)$. As shown in Algorithm 1 (lines 6–7), in each iteration, we select p from the top of the list, and apply trial perturbing moves (a series of "dry runs" that do not actually perturb) to each eligible v-pin in pair p within a predefined small radius r (detailed in Algorithm 2) to find the most efficient move (v^*, δ^*) which means to move v-pin v^* by amount δ^*. Efficiency of a move is defined in terms of the decrease in the model output $-\Delta f(\mathbf{x}) \equiv f(\mathbf{x}_{\text{before}}) - f(\mathbf{x}_{\text{after}})$ and the extra WL $\Delta WL \equiv WL_{\text{after}} - WL_{\text{before}}$ (as an integer).

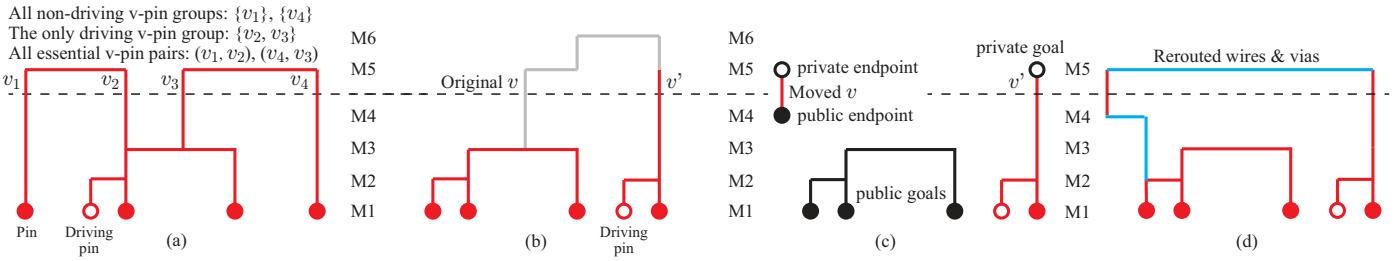


Figure 4: (a) Illustration of terminology. (b–d) Rip up and reroute for v-pin pair (v, v') when v is perturbed. (b) Original wires and vias of the net containing v and v'; the gray segments are to be removed. (c) The new location of v after perturbation is identified. The unconnected parts (including both endpoints of v and rerouting goals) are identified in the public layers (shown in black wires and dots) and private layers (shown in black circles). (d) The unconnected parts are reconnected (in blue).

Algorithm 1: VIA-PERTURBATION ($\mathcal{L}$, R, r, N, W)

Input: $\mathcal{L}$: list of all essential v-pin pairs, R: perturbing direction, which is X for odd split layer and Y for even split layer, r: radius for trial perturbing, N: maximum number of iterations, W: wirelength budget

1 **for** *iter* ← 1 ***to*** N **do**
2 w ← total wirelength of the layout
3 **if** $\mathcal{L}$ *is empty* ***or*** $w \geq W$ **then**
4 **break**
5 **end**
6 **for** p ***in*** $\mathcal{L}$ *in descending order of* $S_{\max}(p)$ **do**
7 $(v^*, \delta^*, \_)$ ← TRIAL-PERTURB (p, R, r, False) `// Algo. 2`
8 **if** $v^* \neq$ *null* **then** `// take the actual move`
9 RIPUP-AND-REROUTE (v^*, R, δ^*, False) `// Algo. 3`
10 Update the feature vector and SHAP values of p.
11 Re-check the eligibility of both v-pins in p, and remove p from $\mathcal{L}$ if neither v-pin is eligible.
12 Re-sort $\mathcal{L}$ by $S_{\max}$.
13 **break** `// only move one v-pin at a time`
14 **else**
15 Remove p from $\mathcal{L}$.
16 **end**
17 **end**
18 **end**

Specifically, to quantify the efficiency of a move, we define its *gain* as

$$gain = \begin{cases} -\Delta f(\mathbf{x})/\Delta WL, & \text{if } \Delta f(\mathbf{x}) < 0 \text{ and } \Delta WL \geq 1 \\ 1 - \Delta f(\mathbf{x}), & \text{if } \Delta f(\mathbf{x}) < 0 \text{ and } \Delta WL \leq 0 \\ 0, & \text{if } \Delta f(\mathbf{x}) \geq 0 \text{ or not feasible} \end{cases}, \quad (2)$$

which prioritizes moves that lead to a decrease in model output at no or low extra cost of WL. The trial perturbing is necessary as it would be difficult to estimate the routing feasibility and extra WL without any trials due to complex layout congestion. After the trial perturbing, if there is no feasible move, we remove pair p from $\mathcal{L}$ (line 15), and proceed to the next v-pin pair in $\mathcal{L}$; if there is any feasible move (lines 8–12), we take the actual move that has the highest gain, update the feature vector and the SHAP values (as in Figure 1), re-check the v-pin eligibility, and go to the next iteration.

Please note, the above gain function depends on the change in model output and is not driven by the SHAP values. However, as explained in the flowchart of ObfusX (Figure 1), valunerable v-pins are identified only using the SHAP values (if the two dominant features are diffVpinX/Y and manhattanVpin). SHAP values are also used to identify the order of obfuscation of the vulnerable v-pins. After obfuscation of each v-pin pair, the feature values will be updated and the next vulnerable pair will be identified. This process may result in a v-pin pair to be obfuscated more than once if it later is found to be vulnerable (e.g., after some other pairs are obfuscated). This overall process results in the SHAP values to be altered after obfuscation, and in particular the SHAP values of v-pins with dominant features (diffVpinX/Y and manhattanVpin) will be significantly reduced after obfuscation. Please refer to example of Figure 3.

5.1.3. Rip-up and Reroute Procedure

To apply a perturbing move to a v-pin v, we rip up and reroute the wires connecting v to the other components. To facilitate the rerouting procedure, we rip up v and all wires connecting to v that do not result in more than two connected components, while not touching any other v-pins, as shown in Figure 4(b). Then we move v to the new location and identify the unconnected parts (i.e. both endpoints of v and the other connected components of the net, referred to as "rerouting goals") in the public and private portions, respectively, as in Figure 4(c). Finally, we use A* search algorithm to reconnect the unconnected parts of the net in the public portion using public layers, and then reconnect for the private portion using private layers, as shown in Figure 4(d). Specifically, the routing graph $G(V, E)$ for A* search is built in three dimensions. The vertices are valid routing grids in all metal layers, and the edges are in x, y and z directions, corresponding to potential wires (in x and y directions) and vias (in z direction) where the routing resources permit. Summarized in Algorithm 3, this rip-up and reroute procedure ensures a feasible route (if possible) and optimizes the WL.

Algorithm 2: TRIAL-PERTURB (p, R, r, $lift$)

Input: p: essential v-pin pair to be perturbed, R: perturbing direction, r: radius for trial perturbing, N: maximum number of iterations, $lift$: whether to apply weight penalties (only used in wire lifting)

Output: v^*: best v-pin to move, δ^*: amount to move, *maxGain*: max possible gain

```
w ← total wirelength of the layout
v* ← null, δ* ← null
maxGain ← 0
for eligible v in v-pin pair p do
    for δ ← −r to r do
        gain ← RIPUP-AND-REROUTE(v, R, δ, lift)
         // move v in R-dir by δ
        if gain > maxGain then
            v* ← v, δ* ← δ
            maxGain ← gain
        end
    end
end
return (v*, δ*, maxGain)
```

Algorithm 3: RIPUP-AND-REROUTE (v, R, δ, $lift$)

```
Rip up v and any wire connecting to v that does not
  result in more than two connected components or touch
  other v-pins. (Figure 4(b)).
Move v in R direction by amount δ.
Identify unconnected parts for rerouting (Figure 4(c)).
Build/update the routing graph, where a vertex = a
  routing grid, an edge = a connection between grids if
  routing resources permit, edge weight = wirelength
if lift then
    Multiply by a large number (e.g. 1000) all edge
      weights for metal wires (not including vias) below
      the split layer
end
reroute using A* search algorithm (Figure 4(d)).
Calculate gain according to (2).
return gain
```

5.2. ObfusX with Wire Lifting

Wire lifting is the second routing-based technique in ObfusX. It moves wires from the public layers to private layers, and therefore creates more v-pins, which can make the attack more difficult.

5.2.1. Wire lifting as rip up and rerouting

To unify our two obfuscation techniques, we propose our wire lifting as a special case of rip-up and rerouting, so that a similar flow to via perturbing can be followed. In wire lifting, instead of exploring the v-pins on the split layers, we explore vias on a *focused layer* that is one or more layers below the split layer. The proposed wire lifting is a two-stage process. In the first stage, we iteratively apply trial rerouting to a via v on the focused layer with the same rip up and reroute procedure to v as in Section 5.1.3, except that (a) to save WL, we do not move the location of v after ripping up, and (b) when rerouting with A* search, we put a higher weight on wires in public layers, so that the use of public wires is discouraged and thus extra v-pins can be created.

5.2.2. Two modes of wire lifting

To make the attack more difficult, wire lifting can be done in either of two ways: (a) to lift wires in a single net, such that newly created v-pins (in the same net) are likely to be perceived by the attacker as "not connected," or (b) to lift wires in two different nets, such that newly created v-pins in different nets are likely to be perceived as "connected." Accordingly, wire lifting in ObfusX can work in two modes: "same net" and "different nets."

In "same net" mode, we try to find the wire that, when lifted above the split layer, would create an essential v-pin pair p' in the same net on the split layer whose model output is low (i.e., it is mistakenly perceived as "not likely to be connected" above split layer), at the cost of no or little wirelength overhead. For each possible essential v-pin pairs p on the focused layer, we follow the trial rerouting procedure to explore the gain of lifting this pair. If the trial rerouting creates a new essential v-pin pair, we evaluate the gain of this move with (2), where $-\Delta f(\mathbf{x}) \equiv f(\mathbf{x}_{\text{before}}) - f(\mathbf{x}_{\text{after}}) = 1 - f(\mathbf{x}_{\text{after}})$, i.e., the model output before the move is defined as 1 because the pair is known to be connected if the vias were not lifted. We record the move and gain in a list of candidate moves, and revert the changes to explore the move with the next essential v-pin pair. After going through all pairs p, we go to the second stage, where we commit the moves to the layout in this list in decreasing order of their gains.

In "different nets" mode, we try to find a pair of vias below the split layer that satisfy all of the following criteria: (a) they belongs to different nets, (b) at least one of the two nets is fully below the split layer, (c) one via is connected to a driving pin, and the other is not, and (d) both vias can be lifted to the split layer. The goal is to create a "deceptive" essential v-pin pair with a high model output (i.e., it is mistakenly perceived as "very likely to be connected" above split layer) with no or little wirelength overhead. The procedure is similar to the "same net" mode, except that (a) in "different nets" mode we explore moves with each *pair* of vias that satisfies the above criteria instead of each single via (each move essentially consists lifting two vias to the split layer); (b) if any of these two vias cannot be lifted to the split layer due to routing infeasibility, the "deceptive" pair cannot be created and therefore this pair would not be considered; and (c) the calculation of gain is slightly different, which is given by

$$gain_{\text{diff}} = \begin{cases} f(\mathbf{x}_{\text{after}})/\Delta WL, & \text{if } f(\mathbf{x}_{\text{after}}) > 0 \text{ and } \Delta WL \geq 1 \\ 1 + f(\mathbf{x}_{\text{after}}), & \text{if } f(\mathbf{x}_{\text{after}}) > 0 \text{ and } \Delta WL \leq 0 \\ 0, & \text{if } \Delta f(\mathbf{x}_{\text{after}}) = 0 \text{ or not feasible} \end{cases} . \quad (3)$$

Compared to (2), all $-\Delta f(\mathbf{x})$ is replaced by $\Delta f(\mathbf{x}) \equiv f(\mathbf{x}_{\text{after}}) - f(\mathbf{x}_{\text{before}}) = f(\mathbf{x}) - 0 = f(\mathbf{x}_{\text{after}})$. This is to reflect that we are dealing with vias in different nets, hence a higher model output $f(\mathbf{x}_{\text{after}})$ is better, as opposed to the case in the "same net" mode. Before lifting, the two vias are known to be not connected, hence $f(\mathbf{x}_{\text{before}})$ being 0.

5.2.3. Lifting different layers of wires

In [11] and [17], wire lifting was performed only for vias on one layer immediate below the split layer. This can be effective when the WL overhead budget is tight. However, limiting the number of layers as such means that wires in nets that only occupy lower layers are never considered to be lifted. Therefore, it is desired to consider more layers of wires to be lifted. In this paper, we consider lifting wires in different layers, from one layer below the split layer down to the bottom-most via layers.

5.2.4. The order of committing moves

There are at least two orders of committing moves. Order A is to commit valid moves as soon as we find a valid move, which is adopted in [17]. In this paper, we propose order B—first store all valid moves in a list and then commit the moves in decreasing order of their gain. Each order has its own pros and cons. With order A, we cannot find the optimal order of exploring (and committing) the moves beforehand, because it is hard to get a good estimate of the WL cost without performing the time-consuming trial routing. In [17], the order to explore is determined by the model output before move, without considering the WL cost. The proposed order B can result in much better tradeoff between obfuscation performance and WL overhead, because Order B considered the WL cost. However, we should note that with order B, the moves are not guaranteed to be committed as stored in the list, because earlier commits may results in changes in routing resources which is not updated in the list. In contrast, with order A, the routing resources are always up-to-date since we explore the next move after committing the previous one, and therefore the trial routing always renders accurate lifting feasibility and routes.

In summary, the procedure of wire lifting is shown as Algorithm 4 where the proposed order B is adopted. Note that in lines 4 and 16, since we do not perturb the location of focused vias, the radius in trial perturbing or rerouting is set to 0 and the direction can be arbitrary.

Algorithm 4: WIRE-LIFTING (l, N, W)

Input: l: the focused layer, N: maximum number of moves, W: wirelength budget

1 $\mathcal{L} \leftarrow$ essential v-pin pairs as if l were the split layer
2 $\mathcal{G}$ = [ ] `// keep track of candidate vias and gains`
3 **for** p ***in*** $\mathcal{L}$ **do**
4 $\quad (v^*, \_, maxGain) \leftarrow$ TRIAL-PERTURB (p, X, 0, True)
5 $\quad$ **if** $v^* \neq null$ **then**
6 $\quad\quad \mathcal{G}$.append((v^*, $maxGain$))
7 $\quad$ **end**
8 **end**
9 Sort $\mathcal{G}$ in decreasing order of $maxGain$
10 $n \leftarrow 0$ `// keep track of number of moves`
11 **for** *(v^*, $maxGain$) in* $\mathcal{G}$ **do**
12 $\quad w \leftarrow$ total wirelength in the layout
13 $\quad$ **if** $n \geq N$ ***or*** $w \geq W$ **then**
14 $\quad\quad$ **break**
15 $\quad$ **end**
16 $\quad$ RIPUP-AND-REROUTE (v^*, X, 0, True) `// commit the moves`
17 $\quad n \leftarrow n + 1$
18 **end**

6. Experimental Results

We obtained the source code of the ML attack from [5], used the `shap` library for Python for SHAP analysis, and implemented all procedures of ObfusX in C++. Experiments were done on a Linux workstation with an Intel 16-core 3.60 GHz CPU and 64 GB memory.

6.1. Via Perturbation with ObfusX

We first show in Table 1 the performance of via perturbation with ObfusX using five designs in ISPD'11 benchmark suite [18] that are also used in [4, 12, 5]. We obtain routed overflow-free designs from [5], to which we apply the proposed SHAP-based via perturbation. As stated in Algorithm 1, parameter r controls the radius of perturbation when calling the trial-perturb function. Higher r results in a higher runtime because it defines a larger space during via perturbation. In our experiments we used $r = 3$ units of the grid size to reach a suitable tradeoff between runtime and quality of solution. This value of r in practice is significantly smaller than the size of the 3D routing grid of the ISPD'11 benchmarks which is in order of 100x100x9. It defines a small search neighborhood compared to the total search space, yet it still allows obtaining a reasonable tradeoff between runtime and solution quality as we demonstrate in our experiments.

We compare the performance and the cost of obfuscation with the via perturbation technique proposed in [5]. This is based on the same ML attack model. Note that the popular network flow attack model [3] takes prohibitively long time to run on these designs and hence is not applicable here. We also do not explicitly compare with a routing congestion-based attack model such as [4] because the work [5] reports results that it is superior to [4].

We use the following metrics to evaluate the performance and the cost of an obfuscation.

- Hit Rate (HR) at X%: For a v-pin v, we first identify the top X% of other v-pins u which have the highest ML model output for essential v-pin pair (v, u). These v-pins are predicted by ML to most likely be the match for v. We call it a "hit" of v if its real matching v-pin is among the v-pins identified above. We then report the average per-

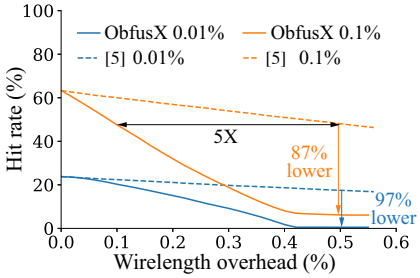


Figure 5: Comparison of tradeoff in HR vs WL in `superblue1`.

centage of hits of all v-pins v in the design. We report this metric with X = 0.01 and 0.1. (As a point of reference, X = 0.1 results in up to 89 v-pins identified on split layer M6, or up to 200 v-pins on split layer M4 in these designs. The total number of v-pins is quite large as reported in the first column of the table.) *A lower HR means better defense.*

- WL overhead (ΔWL%): percentage of increase in WL after the obfuscation. *Lower is better.*
- Perturbed Nets (PN%): number of perturbed nets divided by total number of nets that *contain any v-pin*. *Lower is better.*
- Perturbed V-pins (PV%): number of perturbed v-pins divided by the number of v-pins in the design. *Lower is better.*
- Total runtime of obfuscation using one CPU core (t_{CPU}).

Note that the functions of standard cells are not available in ISPD'11 benchmark. Therefore metrics related to circuit outputs (e.g. Hamming distance (HD), output error rate (OER)) are not applicable.

Several observations can be made from the results in Table 1. **First**, the HR of the ML model for 0.01% and 0.1% v-pin lists drops drastically after obfuscation; for ObfusX it drops from 28% and 67% to 4% and 13%, respectively, better than the HR reductions with in [5]. **Second**, the WL overhead of ObfusX is less than 1/5 of that with [5]. **Third**, with ObfusX, only around 30% of v-pins and 60% of nets (that contain v-pins) are finally perturbed, compared to nearly-all nets and v-pins when perturbed with [5].

To observe the tradeoff between performance and cost of obfuscation, we plot in Figure 5 the curves of HR and WL overhead with ObfusX and [5], respectively. Compared to [5], ObfusX achieves 87% and 97% lower HR in 0.1% and 0.01% v-pin lists, respectively, for the same WL overhead of 0.5%, or is 3–5× more efficient in WL overhead for the same reduction of HR.

6.2. *Wire Lifting with ObfusX*

We show in Table 2 the performance and cost of wire lifting with ObfusX (r = 5 µm) on ISCAS'85 benchmark designs, which are often used in related work, and compare them with [11]. The layouts are obtained from the authors of [11]. The lifting layer are limited to one layer below the split level. The route changes are committed in order A (by model output). Empirically, this is a good setting in general when WL budget is small. Detailed discussion can be found later in Section 6.3.

For this benchmark, we use the network flow attack model [3] which is obtained from the authors. Note that this is *not* a ML-based attack model and is *not* used to build ObfusX. Since the split layer for each design is not explicitly reported in [11], we tried to identify it by matching the number of nets on private layers with the number reported in [11]. ObfusX was applied on six designs for which we were able to identify the split layer, with WL budget equal to the reported WL overhead in [11]. The obfuscated layouts are converted to Verilog and their functional equivalency with original designs is verified with Synopsys Formality. For these designs, we use the following metrics to evaluate the performance and cost of an obfuscation.

- Percentage of Netlist Recovery (PNR) given in [12]: percentage of correctly reconstructed nets. This quantifies how well the attack can recover the whole design. *Lower is better.*
- Output Error Rate (OER): probability that there is any error bit in outputs of the reconstructed circuit. *Higher is better.*
- Hamming Distance (HD) between outputs of the original and the reconstructed circuits. *Closer to 50% is better.*
- WL overhead (ΔWL%): percentage of increase in WL after the obfuscation. *Lower is better.*
- Total runtime of obfuscation using one CPU core (t_{CPU}).

We derive OER and HD from 100,000 runs of Monte Carlo simulations with ModelSim. OER, HD, and the WL overhead of [11] are quoted from [11]. PNR of the original design and [11] are derived by definition, based on the design layouts and the reported numbers in [11]. Although we made our best effort to conduct a fair comparison, we cannot obtain the same version of layout and program that produces the exact results of [11]. (They are similar on average, though.) Instead, we obtain the most recent and publicly available version of layout and program as released by the authors of [11] and run our experiments before and after obfuscation. With this in mind, we keep the results from our experiments and their results from [11], and compare the *changes* before and after obfuscation in [11] and in our experiments.

As can be seen in Table 2, with a computing time of less than five minutes, ObfusX can reach 100% for OER, and achieve better obfuscation in the reduction of PNR (11% vs 5% on average, or 2.2× better) and the increase in HD (24.6% vs 18.6% on average, or 32% better), with the same or less WL overhead compared to [11]. Note that the reported results of [11] come from a (best) combination of three obfuscation techniques including wire lifting and via perturbation for matching and non-matching v-pins, whereas in our results wire lifting is applied alone. In fact, our wire lifting and via perturbation techniques

Table 1: Results of via perturbation with ObfusX on the ISPD'11 benchmark suite

Split layer	Design (#v-pins)	No obfuscation HR @ 0.01% / 0.1%	[5] HR @ 0.01% / 0.1%	[5] ΔWL%	[5] PN% / PV%	[5] t_{CPU} (h)	ObfusX HR @ 0.01% / 0.1%	ObfusX ΔWL%	ObfusX PN% / PV%	ObfusX t_{CPU} (h)
M6	sb1 (44486)	23.79 / 63.33	2.19 / 11.58	3.03	99.83 / 99.58	3.86	0.52 / 6.12	0.55	66.57 / 36.01	3.28
	sb5 (60034)	29.47 / 63.96	5.75 / 20.38	4.09	96.81 / 91.75	7.13	4.34 / 15.46	0.67	55.62 / 30.08	5.30
	sb10 (89846)	31.84 / 64.34	10.24 / 28.31	4.52	92.45 / 79.77	7.75	9.37 / 23.93	0.71	46.49 / 23.96	8.05
	sb12 (80816)	33.01 / 75.58	8.23 / 24.78	3.31	97.70 / 90.12	6.46	4.32 / 11.67	0.64	73.87 / 37.12	5.45
	sb18 (36026)	20.06 / 66.11	4.27 / 16.55	2.64	98.91 / 94.35	2.88	2.16 / 8.68	0.67	63.02 / 34.27	2.06
	Average	27.63 / 66.66	6.14 / 20.32	3.52	97.14 / 91.11	5.62	4.14 / 13.17	0.65	61.11 / 32.29	4.83
M4	sb1 (150510)	49.82 / 68.33	6.46 / 25.37	9.50	99.79 / 93.91	9.00	1.70 / 24.08	2.14	65.23 / 35.26	18.90
	sb5 (179844)	38.78 / 60.40	7.54 / 23.84	9.86	96.94 / 87.87	11.48	3.03 / 23.35	1.87	51.43 / 28.09	18.41
	sb10 (200896)	33.50 / 60.21	13.16 / 37.36	8.53	91.38 / 73.21	15.05	9.81 / 36.54	1.31	38.81 / 19.55	17.19
	sb12 (173294)	47.07 / 71.52	9.01 / 22.40	7.61	98.61 / 92.32	13.48	4.42 / 17.39	1.12	65.32 / 32.81	18.09
	sb18 (86658)	29.83 / 59.89	5.15 / 17.89	6.43	99.37 / 95.29	4.26	1.87 / 10.95	1.53	57.00 / 30.80	7.18
	Average	39.80 / 64.07	8.26 / 25.37	8.39	97.22 / 88.52	10.65	4.17 / 22.46	1.59	55.56 / 29.30	15.95

Table 2: Results of wire lifting with ObfusX with the ISCAS'85 benchmark suite

Design	#Nets	No obfuscation, from [11] PNR%	OER%	HD%	Obfuscated, quoted from [11] PNR%	OER%	HD%	ΔWL%
c880	252	100.0	0.0	0.0	91.7	99.9	18.0	4.3
c2670	607	95.8	99.9	7.0	87.1	100.0	14.0	4.4
c3540	638	97.2	95.4	18.2	93.5	100.0	33.4	2.5
c5315	997	98.7	98.7	4.3	95.0	100.0	18.1	1.7
c6288	1921	99.8	36.8	3.0	98.6	100.0	42.1	1.8
c7552	1041	99.6	69.5	1.6	95.3	100.0	20.3	2.2
Avg.		98.5	66.7	5.7	93.5	100.0	24.3	2.8
Comparing to "No obfus."					-5.0	+33.3	+18.6	

Design	#Nets	No obfuscation, from our experiments PNR%	OER%	HD%	Obfuscated with ObfusX PNR%	OER%	HD%	ΔWL%	t_{CPU} (min)
c880	252	99.2	50.0	1.9	**85.3**	**100.0**	**24.6**	**3.4**	1.2
c2670	607	95.8	100.0	5.8	**77.9**	100.0	**23.3**	**3.2**	2.4
c3540	638	98.6	79.2	9.5	**87.0**	100.0	**37.4**	2.5	4.7
c5315	997	97.5	99.5	10.4	**89.6**	100.0	**23.7**	1.7	4.4
c6288	1921	100.0	0.0	0.0	**96.8**	100.0	**46.6**	1.8	3.5
c7552	1041	99.1	98.9	4.5	**87.0**	100.0	**24.4**	2.2	4.7
Avg.		98.4	71.3	5.4	**87.3**	**100.0**	**30.0**	**2.5**	3.5
Comparing to "No obfus."					**-11.1**	+28.7	**+24.6**		

are orthogonal to each other. Therefore, they may be combined for potentially better performance.

We were not able to make a fair comparison with another related work [12] because the original layouts of [12] are likely to be very different from ours and were not made available. (The layouts in [12] are generated using all 10 metal layers, whereas our layouts from [11] only occupy 5–9 lower metal layers.)

In summary, for obfuscation with via perturbation, ObfusX is able to achieve a lower hit rate (indicating better obfuscation) while perturbing significantly fewer nets and vias in the design, with significantly lower wirelength. When the same wirelength limit is imposed during wire lifting, ObfusX performs significantly better in performance metrics (PNR and HD with equally good OER).

6.3. Comparison of wire lifting with different settings

Using the same six ISCAS'85 designs, we compare the results of wire lifting with different settings as described in Section 5.2, including

- lifting one, two, and three layers of vias below the split layer (denoted `1L`, `2L` and `3L`, respectively);
- committing in order A (by model output, denoted `mo`) and order B (by gain, denoted `gain`), in "same net" mode;
- lifting in "same net" mode (denoted `gain`) and "different nets" mode (denoted `diff`), committed in order B.

The naming of settings is a combination of (`1L`, `2L`, `3L`) and (`mo`, `gain`, `diff`). Please note the work [17] corresponds to the `1Lmo` case.

To ease comparison, we use the following metric to quantize the obfuscation performance.

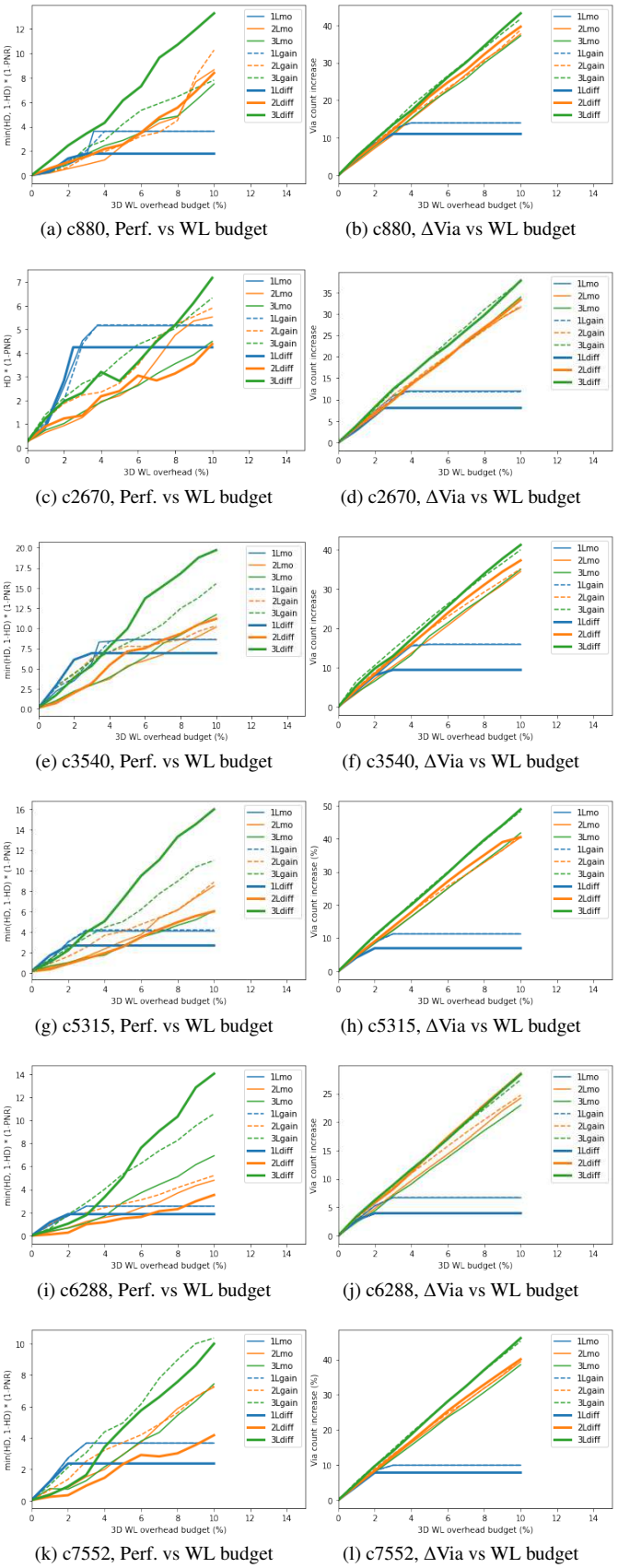
$$Perf = (1 - PNR) \cdot \min(HD, 1 - HD). \quad (4)$$

This unified performance metric has a higher (better) value when PNR is lower and HD is closer to 50%. We do not include

OER because it is very close to 100% most of the time. We plot the performance and increase in via count verses the WL overhead budget (in percentage) for each design in Figure 6.

We can make the following observations from these results.

1. For `1L*` settings (* serves as a wildcard), the performance is good for small WL budget—by limiting the layer of lifted wires, it tends to obfuscate many different nets with small WL overhead. However, the performance and via count saturates after a certain WL overhead budget—these settings only discover nets that extend to the metal layer immediately below the split layer, thus the lifting options exhaust earlier than `2L*` and `3L*` settings. Specifically, the `1Ldiff` setting saturates even earlier than other `1L*` settings because in "different nets" mode, a move involves two vias in different nets, and *both* vias need to be feasible to reroute above the split layer in order to be considered a valid move, whereas in the "same net" mode, only one via is involved in a move. This stricter condition further reduces the number of potential lifting options.
2. In terms of the tradeoff between obfuscation performance and WL overhead (as the slopes in Figure 6(a)(c)(e)(g)(i)(k)), `2L*` settings are generally worse than others. This is because the focused layer (two layers below the split layer) has the same preferred routing direction as the split layer. Vias on the focused layer are more likely to share similar properties to the vias on the split layer (for example, one of the coordinates of via locations could be the same). This can make the attack relatively easier than other focused layers (as in `1L*` and `3L*` settings).
3. The `*gain` settings have better tradeoffs between obfuscation performance and WL overhead than `*mo` settings. These two groups differ in the order of committing changes, as detailed in 5.2.2. As expected, `*gain` settings win as they prioritize moves with higher gain, which implicitly consider the WL cost whereas `*mo` settings do not.
4. The `*diff` settings generally have better tradeoffs between obfuscation performance and WL overhead than `*gain` settings. This two groups correspond to the "different nets" mode and "same net" mode, respectively. The better tradeoffs are attributed to the much more lifting options to explore from pairs of vias from different nets than from the same net. Therefore, the best move in the `*diff` mode can be potentially superior to that in the `*gain`.
5. Moving to Figure 6(b)(d)(f)(h)(j)(l), the via count grows almost linearly with the WL overhead budgets, with similar slopes among different settings (except for `1L*` settings where the growth terminates after a point due to limited valid options to lift (see observation 1). This is because our WL estimation includes the estimated length of vias, therefore the number of vias cannot grow unlimitedly given a WL overhead budget. The linearity of growing also means the proportion of WL in metal layers and vias remains steady regardless of the WL overhead budget.
6. A closer look at the slopes in via vs WL overhead budget plots reveals that settings `3Ldiff` and `3Lgain` leads to



(a) c880, Perf. vs WL budget (b) c880, ΔVia vs WL budget

(c) c2670, Perf. vs WL budget (d) c2670, ΔVia vs WL budget

(e) c3540, Perf. vs WL budget (f) c3540, ΔVia vs WL budget

(g) c5315, Perf. vs WL budget (h) c5315, ΔVia vs WL budget

(i) c6288, Perf. vs WL budget (j) c6288, ΔVia vs WL budget

(k) c7552, Perf. vs WL budget (l) c7552, ΔVia vs WL budget

Figure 6: Performance (Perf.) and increase in via count (ΔVia, in percentage) verses WL budget (%) of wire lifting with ObfusX for six designs in ISCAS'85 benchmark suite.

generally more new vias on split layer per unit WL overhead than `1L*` and `2L*`, and `3Lmo` settings. This is because with the most nets being considered in `3L` settings, we has the most valid moves to consider. Also when we commit by gain as in both `*diff` and `*gain` settings, we commit better moves as we take the WL cost into consideration.

7. Conclusions

We presented ObfusX, a routing obfuscator for split manufacturing which incorporated SHAP-based analysis to explain a machine learning attack. The unique benefits of ObfusX are in its ability to identify the best candidate nets for obfuscation together with the layout features which make them most vulnerable when subjected to an attack. As a result, it achieves better performance than prior work while perturbing significantly fewer nets and with significantly lower wirelength during via perturbation. It also achieves significantly better performance than prior work if the same wirelength limit was imposed during wire lifting.

8. Acknowledgments

This research was supported by National Science Foundation Grant No. 1812600 and by Semiconductor Research Corporation Grant No. 2845.

References

[1] J. Rajendran, O. Sinanoglu, R. Karri, Is split manufacturing secure?, in: DATE, 2013, pp. 1259–1264.

[2] K. Vaidyanathan, B. P. Das, E. Sumbul, R. Liu, L. Pileggi, Building trusted ICs using split fabrication, in: 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 2014, pp. 1–6.

[3] Y. Wang, P. Chen, J. Hu, G. Li, J. Rajendran, The cat and mouse in split manufacturing, IEEE Trans. VLSI Syst. 26 (2018) 805–817.

[4] J. Magaña, D. Shi, A. Davoodi, Are proximity attacks a threat to the security of split manufacturing of integrated circuits?, IEEE Trans. VLSI Syst. 25 (2017) 3406–3419.

[5] W. Zeng, B. Zhang, A. Davoodi, Analysis of security of split manufacturing using machine learning, IEEE Trans. VLSI Syst. 27 (2019) 2767–2780.

[6] Y. Wang, T. Cao, J. Hu, J. Rajendran, Front-end-of-line attacks in split manufacturing, in: ICCAD, 2017, pp. 1–8.

[7] W. Xu, L. Feng, J. Rajendran, J. Hu, Layout recognition attacks on split manufacturing, in: ASPDAC, 2019, pp. 45–50.

[8] H. Li, S. Patnaik, M. Ashraf, H. Yang, J. Knechtel, B. Yu, O. Sinanoglu, E. F. Y. Young, Deep learning analysis for split manufactured layouts with routing perturbation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2020).

[9] K. Xiao, D. Forte, M. M. Tehranipoor, Efficient and secure split manufacturing via obfuscated built-in self-authentication, in: HOST, 2015, pp. 14–19.

[10] A. Sengupta, S. Patnaik, J. Knechtel, M. Ashraf, S. Garg, O. Sinanoglu, Rethinking split manufacturing: An information-theoretic approach with secure layout techniques, in: 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017, pp. 329–326.

[11] Y. Wang, P. Chen, J. Hu, J. Rajendran, Routing perturbation for enhanced security in split manufacturing, in: ASPDAC, 2017, pp. 605–610.

[12] S. Patnaik, J. Knechtel, M. Ashraf, O. Sinanoglu, Concerted wire lifting: Enabling secure and cost-effective split manufacturing, in: ASPDAC, 2018, pp. 251–258.

[13] S. Patnaik, M. Ashraf, J. Knechtel, O. Sinanoglu, Raise your game for split manufacturing: Restoring the true functionality through BEOL, in: DAC, 2018, pp. 140:1–140:6.

[14] S. M. Lundberg, S.-I. Lee, A unified approach to interpreting model predictions, in: NeurIPS, 2017, pp. 4765–4774.

[15] E. Frank, M. A. Hall, I. H. Witten, The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, 4 edition, 2016.

[16] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, S.-I. Lee, From local explanations to global understanding with explainable AI for trees, Nature Machine Intelligence 2 (2020) 56–67.

[17] W. Zeng, A. Davoodi, R. O. Topaloglu, ObfusX: Routing obfuscation with explanatory analysis of a machine learning attack, in: Proceedings of the 26th Asia and South Pacific Design Automation Conference, 2021, pp. 548–554.

[18] N. Viswanathan, C. J. Alpert, C. C. N. Sze, Z. Li, G.-J. Nam, J. A. Roy, The ISPD-2011 routability-driven placement contest and benchmark suite, in: ISPD, 2011, pp. 141–146.