Analysis of Graph-based Partitioning Algorithms and Partitioning Metrics for Hardware Reverse Engineering

Selina Weber, Johanna Baehr, Alexander Hepp, Georg Sigl

Analysis of Graph-based Partitioning Algorithms and Partitioning Metrics for Hardware Reverse Engineering *

Selina Weber[1], Johanna Baehr[1], Alexander Hepp[1], and Georg Sigl[12]

[1] Chair of Security in Information Technology
TUM School of Computation, Information and Technology
Technical University of Munich, Germany
{Selina.M.Weber, Johanna.Baehr, Alex.Hepp, Sigl}@tum.de
[2] Fraunhofer Institute for Applied and Integrated Security (AISEC), Germany

Abstract

With modern Integrated Circuit (IC) fabrication taking place offshore and with third-party companies, hardware reverse engineering has become an effective method to ensure the security of chips. Recently, it has gained more and more attention to counteract the threats of Intellectual Property (IP) theft, overproduction, and Hardware Trojan (HT) insertion. However, to reverse engineer real-world ICs, methods must scale to millions of logic gates. This is also true for the final step in hardware reverse engineering: netlist abstraction. Here, a divide and conquer approach has become necessary, where the gate-level netlist is divided into smaller partitions, which are then identified separately. This work introduces several graph-based methods for netlist partitioning, which are faster, more accurate, more flexible, and require less information about the design than current solutions. The algorithmic efficiency of these methods is compared using theoretic analysis and experimental assessment. These experiments also evaluate the correctness of the partitioning methods for small and large netlists, using several evaluation metrics. Furthermore, this work analyses these metrics' behavior for different types of netlists and discusses why a single metric is insufficient to evaluate partitioning methods correctly.

1 Introduction

In a world of globalized Integrated Circuit (IC) supply chains, hardware assurance is a crucial step towards secure and trustworthy devices. Devices must be protected against malicious intruders, who might exploit a lack of control to manipulate the devices for their own purposes. Potential threats include insertion of Hardware Trojans (HTs), Intellectual Property (IP) piracy, overbuilding, and counterfeiting [1], [2].

Both an intruder trying to understand a design to launch a targeted attack, and a defender trying to find and understand possible attacks must use Reverse Engineering (RE) to gain additional knowledge about a target design. RE can thus be seen both as a threat to defend against, as well as a valuable tool for IC trust. In any case, it is important to understand if and how RE can be applied to a given design, either to evaluate an attacker's capabilities and the success of defense strategies or enable successful inspection of a design for IP theft and HTs. In this work, we focus on one important step in RE: analyzing the flat netlist, which is the result of hardware synthesis into logic gates and connecting wires.

In netlist reverse engineering, divide and conquer based approaches are necessary to handle the large number of logic gates on an IC. Many of the approaches to identify the functionality

*This work was partly funded by the German Ministry of Education and Research in the project RESEC under Grant No.: 16KIS1009.

of a design require a golden model, which is often not available for an entire IC. Instead, golden models are often available only for functional submodules of the design. Furthermore, many approaches simply do not scale well for many million gates. Thus, methods to partition designs are required to successfully reverse engineer ICs. These methods must be efficient, scalable, and provide a result that is useful to subsequently identify the functionality of the design.

These methods can be grouped into two classes: data-path based [3]–[5] and graph-based [6]–[9]. Methods in the first class partition netlists by identifying the flow of data through the netlist, described by connected data-words. Partitions are then carved out between such data-words, to form functional submodules of the netlist. However, it can be difficult to identify which combination of data-words should be used as the inputs and outputs of a partition.

Methods in the second class perceive the netlist as a graph, and use graph-based methods to identify clusters, which correspond to functional submodules. This is logical from a design point of view. Clustering methods have been used for layouting and placement in Electronic Design Automation (EDA) tools for many years, as gates belonging to functional submodules should be placed near each other within the layout [6]. Consequentially, methods leveraging similar information can assist in the RE process. Additionally, core-based designs remain common, where functional submodules are highly connected, with only few connections to other submodules [6]. These structures are ideal for identification with graph-based methods.

In this work, we extend previous work on graph-based netlist partitioning methods, as well as continue and expand on the discussion on what constitutes a good partition, and how to fairly measure this. We will first introduce our netlist definition in section 2. In section 3, we will introduce a number of graph-based partitioning algorithms. Each of the previously published works on netlist partitioning uses a different metric to evaluate their partitioning results, which makes a fair and simple comparison between methods difficult. Thus, section 4 will discuss a number of metrics. Their behavior for different types of results, as well as a general analysis of the results for a number of designs, is presented in section 5. We include a run-time and complexity analysis, as well as an in-detail comparison against the published partitioning methods. Finally, we will provide an outlook on the future of netlist partitioning in section 6.

2 Netlist Definitions

In this work, a design netlist is perceived as hypergraph $\mathcal{D} = (\mathcal{M}, \mathcal{N})$, with the set of modules (i.e. standard gates) $\mu \in \mathcal{M}$ and the nets $\nu \in \mathcal{N}$ connecting the modules. All modules connected to the net ν are in the set $\mathcal{M}_\nu$. A hierarchical design netlist can be described by a tree $\mathcal{H}$ with nodes $\mathcal{H}_i^l$ on a hierarchy level l starting with $l = 0$ describing the complete design. Each hierarchical node in the tree instantiates at least two subnodes from higher levels, i.e. $H_i^l = \{H_i^{l'} | l_{max} \geqslant l' > l \geqslant 0\}$ and $|H_i^l| > 0$. Leaf nodes $H_i^{l_{max}}$ are in the highest level l_{max} of the tree and are identical to the modules μ, i.e. $\forall i : H_i^{l_{max}} \in \mathcal{M}$.

$\bar{H}_i^l = \bigcup_{H_i^{l'} \in H_i^l} \bar{H}_i^{l'}$, with $\bar{H}_i^{l_{max}} = H_i^{l_{max}}$, is defined to be the flattened hierarchy, i.e. the set of μ belonging to the hierarchy node itself or to one of the subnodes from higher levels.

For each $\bar{H}_i^l$, we define a directed graph $\bar{G}_i^l = (\bar{V}_i^l, \bar{E}_i^l)$ such that each node $\bar{v}$ is a module $\mu \in \bar{H}_i^l$. An edge $e \in \bar{E}$ exists between two modules $\mu_1, \mu_2 \in \bar{H}_i^l$ if there is a net $\nu \in \mathcal{N}$ in the design netlist with $\mu_1 \in \mathcal{M}_\nu \wedge \mu_2 \in \mathcal{M}_\nu$. As net names and cell names are arbitrary and should not influence the partitioning, only the identifier of μ is kept as an attribute of v. In this work, we do not utilize placement information, thus the edges e have unit length. Another graph $G_i^l = (V_i^l, E_i^l)$ is defined, such that each node v is a module $\mu \in \{\mu | \mu \in H_i^l\}$, i.e. this

graph comprises only the nodes describing the immediate function of this hierarchy node. The graph $\bar{G}_0^0$, in which no hierarchy is given, is the typical flat netlist input data for netlist reverse engineering. In this work, we refer to the set of all G_i^l and $\bar{G}_i^l$ in one design as L.

Graph Partitioning Graph partitioning is performed to receive a ground truth to evaluate graph clustering algorithms. We use the available hierarchical description and generate the G_i^l and $\bar{G}_i^l$ for all nodes in the hierarchy tree, to receive the set L for each design under test.

Graph Clustering The ultimate goal of RE for netlist partitioning is the hierarchy tree $\mathcal{H}$ representing the golden partitioning of the hardware design. By utilizing graph clustering algorithms, we try to achieve hierarchy trees $\hat{\mathcal{H}}$ that are as close to as possible to the true $\mathcal{H}$. The focus lies on unsupervised clustering algorithms, i.e. algorithms that do not depend on many parameters and do not partition towards a predefined number of hierarchy nodes.

Each algorithm (with given parametrization) generates many graphs $\hat{\bar{G}}_i^l$ of $\bar{G}_0^0$, collected into a set P_{algo} per design.

2.1 Discussion: Ground Truth Definition

In graph theory, there exists a plethora of algorithms to partition a graph. All seek to group nodes within a graph into groups which most closely resemble the ground truth. This ground truth, however, differs strongly depending on the application, and may not even be clearly defined. For netlist RE, the goal is to create understanding of the design. This also applies for the partitioning step. Thus, any partitioning which increases the understanding of the design is a good partitioning.

Graph partitioning often considers two cases: exact partitions and overlapping partitions. For exact partitions, each node in the graph is assigned to a single partition. For overlapping partitions, each node can be assigned to one or more partitions. Netlists are very similar, in that they can have multiple levels of hierarchy. An IC may contain a cryptographic hardware accelerator and a Central Processing Unit (CPU), both of which in turn consists of smaller modules. This issue has been previously discussed by [5], who propose to use data-path based methods to describe the ground truth as closely as possible.

However, we believe that this view is too narrow-minded, especially as designs exist that cannot easily be described only through their data-path. Instead, we believe that there exists no clear rule regarding which level of hierarchy is best to create understanding of the design. However, by considering the next step after netlist RE, the identification of the functionality of the design, some insight into what is a good level of hierarchy can be gained. Large partitions will not allow for a good comparison against a golden model, as a similar module will generally not be contained in a golden model library. An example could be a partition consisting of an entire CPU, for which only parts may be previously known. On the other hand, small partitions provide less knowledge about the design when they are functionally identified, and may be more difficult to join to create a big picture. When partitions become too small, they may no longer represent a known functional submodule, and become only a number of connected logic gates.

In the best case, for verification of new partitioning methods, there exists a ground truth, which is composed of the functional submodules of a design, as intended by an intelligent designer. This assumes that the designer created a design in such a way that functional submodules can be identified and separated. These modules are more likely to be known in the golden model library, and gates belonging to these modules can generally be extracted from the netlist after synthesis, if the Register Transfer Level (RTL) description is known. The

extraction of this ground truth is however not always trivial. If each functional submodule is described in its own module, then, during synthesis, gates which belong to this module can be recovered. However, optimizations and the multi-hierarchical nature of netlists can make this process difficult. Without a structured RTL representation this becomes even more difficult. Thus, good RTL design can lead to easier evaluation of partitioning methods.

In the evaluation in this paper, we assume that the ground truth can be extracted as described above. Furthermore, since we evaluate exact partitioning methods, we create our ground truth to also only represent an exact partitioning.

3 Graph-based Algorithms for Netlist Partitioning

This section outlines the graph partitioning algorithms that were used in our experiments.

3.1 Louvain

The *Louvain* method was proposed by Blondel et al. [10] for community detection and focuses on optimized performance and scalability in comparison to other graph clustering algorithms. For initialization, each node in the graph is assigned to a cluster, consisting only of this one node. After that, two phases are performed repeatedly: the clusters are merged with another cluster in the vicinity, and the gain of these new clusters is evaluated, until a local maximum for the current node arrangement is achieved. This is performed until there are no more changes to the clustering's structure that can improve the clustering. This local maximum is determined utilizing the modularity:

$$H_{\text{mod}} = \frac{1}{2m} \sum_{c} (e_c - \gamma \frac{K_C^2}{2m}) \tag{1}$$

in which $\gamma > 0$ is the resolution parameter, e_c is the number of edges in a community c, m is the number of edges in a network, and $\frac{K_C^2}{2m}$ is the expected number of edges in the network with K_c being the sum of the degrees of the nodes in a community c. Additional abstraction can be gained by condensing all nodes in the resulting $\hat{\hat{G}}_i^l$s to a single node in a new graph, and performing the *Louvain* method again on this new graph, forming a higher-level partitioning.

Note that the result of this algorithm depends on the order the nodes are accessed in the first step. This leads to different clusterings for several runs because of the randomness in this phase.

3.2 Leiden-Mod

The *Leiden-Mod* algorithm was published in 2019 by Traag et al. [11] as a successor for the *Louvain* method, which struggled with disconnected or poorly connected clusters. Like *Louvain*, *Leiden-Mod* also uses modularity as the quality function. The initialization is done once and is the same as for *Louvain*. Then, the nodes are moved locally to new clusters. This temporary clustering is refined, before a new network is formed with the changes. These steps are repeated until no more improvement of the modularity can be made. A comparison of the two algorithms shows that for *Leiden-Mod*, the amount of poorly connected communities decreases over time while they generally increase for *Louvain*.

3.3 Leiden–CPM

To overcome the scalability limitations of modularity and resolution, the Constant Potts Model (CPM) is introduced as a quality function for the *Leiden-CPM* method, replacing the modularity.

$$H_{\text{CPM}} = \sum_{c}(e_c - \gamma(\frac{n_c}{2})) \tag{2}$$

in which e_c is the number of edges in a community c, $\gamma > 0$ is the resolution parameter, and n_c is the number of nodes in a community c [11].

The resolution parameter γ influences the number of communities in the clustering. More clusters are found for a higher resolution, and a lower resolution leads to fewer clusters. This method is particularly useful for larger graphs, as is scales well over many millions of nodes.

3.4 MCL

The *Markov Cluster (MCL)* algorithm focuses on the mathematical theory of clustering algorithms for graphs [12]. The algorithm is commonly used in bio-informatics, which requires the analysis of extremely large graphs. It can be configured using the inflation parameter, which affects the granularity of the final clustering. The algorithm is based on the simulations of flows within the graph. It uses the discrete Markov process to apply expansion and inflation to the graph to form communities.

4 Metrics for Partitioning for Hardware Reverse Engineering

Previous work has often concentrated on using a single metric to evaluate the quality of netlist partitions, particularly when evaluating new partitioning methods. Commonly used metrics are the normalized mutual information score (NMI) score [5], accuracy [7], [8], precision, recall or the F1 score [9], [13], while some methods do not use any metrics at all, and instead present their results without further analysis [3], [4], [6].

We instead propose the use of more than a single metric, as single metrics scores can be biased towards a certain type of netlists and show misleading results (see 5.4). The use of several metrics allows us to analyze the score behavior, which provides insight on the correctness of the cluster number and node distribution of the design.

There are two types scores that give information about a clustering. Internal scores of a graph, also called fitness scores, give information about the characteristics of the clustering itself, such as the average distance of nodes in a cluster. External scores are used to compare different graph clusterings, as long as they cover the same nodes. In this work, we use such external scores as metrics to calculate how similar the clustering is to the partitioning. We evaluate the commonly used metrics: adjusted mutual information (AMI), normalized mutual information score (NMI), adjusted Rand Index (ARI), F1, and normalized F1 score (nF1). Each of these provide a score between $[0:1]$, with 1 indicating an exact match between clustering and partitioning, and 0 indicating no match between them. The definition of good score strongly depends on the use-case, and will be discussed in section 5.3.

AMI and NMI are based on mutual information

$$MI(X,Y) = H(X) - H(X|Y), \tag{3}$$

in which $H(X)$ is the entropy of clustering X and $H(Y|X)$ is the conditional entropy of clustering X given clustering Y.

From this, the adjusted mutual information is defined as

$$AMI(X,Y) = \frac{MI(X;Y) - E\{MI(X,Y)\}}{max\{H(X), H(Y) - E\{MI(X,Y)\}\}}, \tag{4}$$

in which $E\{MI(X,Y)\}$ is the expected mutual information.

The normalization of the mutual information is calculated using the joint entropy $H(X,Y)$ of X and Y

$$NMI = \frac{H(X) + H(Y) - H(X,Y)}{(H(X) + H(Y))/2}, \tag{5}$$

The Rand Index can be calculated according to

$$RI = \frac{a+d}{a+b+c+d} \tag{6}$$

in which a refers to the number of pairs that are in the same cluster in the partitioning, and the clustering, b is the number of pairs in different clusters in the partitioning, but the same cluster in the clustering, c describes the pairs in same clusters of the partitioning, but in different clusters in the clustering, and d counts all pairs that are in different clusters for both the partitioning and the clustering.

If the RI is adjusted by the amount of grouping of elements that happens by chance, the ARI can be defined, with the expected value $E\{RI\}$ and the value $Max(RI)$ for scaling, as

$$ARI = \frac{RI - E\{RI\}}{Max(RI) - E\{RI\}}. \tag{7}$$

The F1 clustering score [14] is built on the idea that each cluster can be matched to the best fitting partition by identifying the ground truth partition for each node and performing majority voting for each cluster. Thus, it is possible to count the number of nodes in each cluster that belong to the matched partition and to, calculate precision and recall using the number of `true positive` (tp), `false positive` (tp), `true negatives` (tn) and `false negatives` (fn) nodes

$$precision = \frac{tp}{tp\ +\ fp}, \quad recall = \frac{tp}{tp\ +\ fn}, \tag{8}$$

The F1 score is defined as the harmonic mean of precision and recall is calculated for each cluster

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{9}$$

and the F1 clustering score is received by averaging the F1 for each cluster. The nF1 is the F1 clustering score normalized by the amount of partitions matched by at least one cluster and the amount of redundantly matched partitions [15].

Accuracy, also commonly used to evaluate partitions, can also be defined using these values as

$$accuracy = \frac{tp+tn}{tp\ +fp+fp+fn}. \tag{10}$$

Table 1: Design Characteristics and Run-time

	Design Characteristics		Runtimes [s]			
	#Gates	#Partitions	Louvain	Leiden-Mod	Leiden-CPM	MCL
present	1,361	5	5	5	6	5
BLAKE	22,538	8	115	13	5	57
AES	32,638	43	53	10	10	239
i2c	868	3	3	4	3	4
altor	12,952	8	30	5	4	54
aquarius	22,726	14	54	7	6	100
ethmac	59,908	66	2,069	11	12	1,104
FPU	64,232	7	187	9	8	261
aquarius-mem	683,003	15	7,200	93	539	-

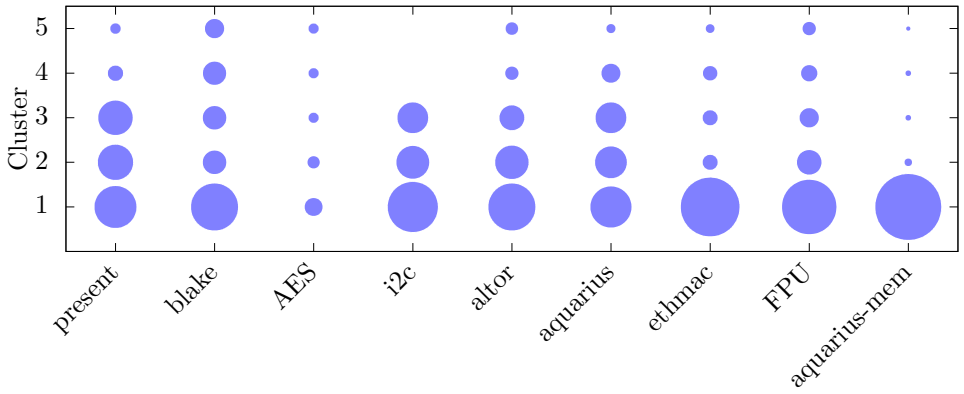


Figure 1: Node Distribution for five largest clusters per Design

5 Results

5.1 Design Overview

To fairly evaluate our methods, we have chosen a number of real-life designs, with a focus on cryptographic cores and components of System on Chips (SoCs). Implementations stem from opencores [16] and secworks [17], and are synthesized using an open source tool chain and an open source technology node. An overview of the design size and the number of partitions is given in Table 1. The smallest design is the `i2c` design, while the largest is the `aquarius-mem` CPU. This design has been included twice, the larger version includes a large memory block, to demonstrate the scalability of our approach. Furthermore, as the node distribution has a strong effect on metric behavior, we also include an overview for the five largest clusters of each design in Figure 1, where the area of each circle represents the relative size of the cluster.

5.2 Clustering Parameter Choice

Both the *Louvain* and *Leiden-Mod* algorithms are unsupervised methods to create graph partitions, requiring no parameters, and no knowledge regarding the design to function. However, *Leiden-CPM* and *MCL* both require a single parameter, which, while it does not control the

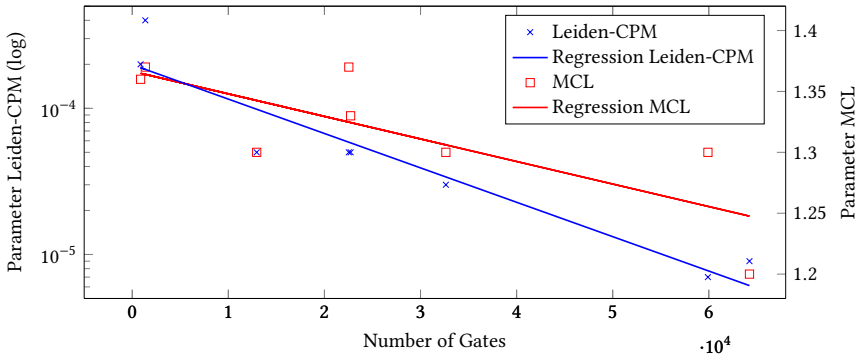


Figure 2: Parameter Choices for *Leiden-CPM* and *MCL*

number of the resulting clusters directly, does control the granularity or resolution of the clusters. Thus, there is correlation between the parameter and the number of clusters. To better understand how to choose this value for both algorithms for different designs, we evaluated the cluster quality for a number of parameters: $I = [1.20 : 1.40]$ for *MCL*, with a total of 8 parameters choices, and $\gamma = [1 \times 10^{-7} : 2 \times 10^{-4}]$ for *Leiden-CPM*, with a total of 16 parameter choices. As the *Louvain* algorithm can output slightly different results, depending on a random seed, this method was also done ten separate times.

The correlation of the parameter choice for the best quality of results against the design size is shown in Figure 2, including a linear regression for both methods. Note the logarithmic scale for *Leiden-CPM*. While the correlation is stronger for *Leiden-CPM*, both methods do show a clear correlation. This information can be leveraged to choose a meaningful parameter set for an unknown design.

5.3 Metric Values for Successful Partitioning

To understand what constitutes a good, a sufficient or a bad partitioning result, the use-case must be considered. As previously mentioned, netlist RE is generally carried out for different purposes: HT insertion, IP piracy, but also for identification of patent violations, and for HT detection. Each of these use-cases provides a different amount of knowledge about the design, but also requires a different quality of partitioning result [18]. Netlist RE in attack scenarios generally will supply no information about the design to be attacked, other than design size, while netlist RE for protection allows for partition number, relative size and even functionality to be used when evaluating a netlist.

Depending on how the next step in netlist RE is carried out, and how error tolerant this method is, a partitioning can be sufficient even without a perfect match between the ground truth and the resulting clusters [19]. However, the next step may instead require an exact partition, as is the case when carrying out a Satisfiability (SAT)-based attack on logic locked netlists. Without prior knowledge of the design, this can be very difficult to achieve [13]. In general, a metric score of > 0.8 is considered good in most scenarios, while a score < 0.6 will likely no longer provide sufficient information about the design to carry out useful netlist RE.

5.4 Analysis of Metric Behavior

The distribution of nodes and the amount of clusters that are formed during the clustering are essential information to judge a clustering's quality. The proposed score based metrics evaluate

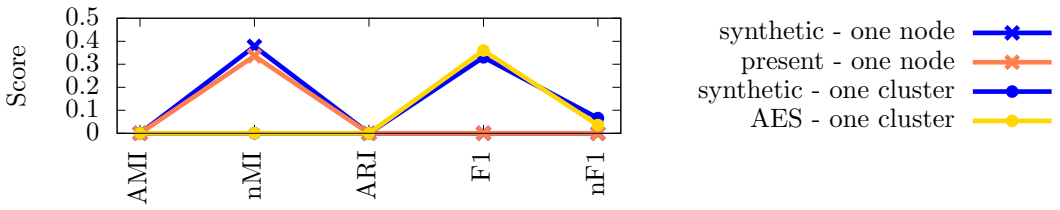


Figure 3: Metric Behavior for one-cluster clustering and one-node clustering

the similarity between clustering and partition, but due to mathematical conditions, the scores vary with the amount of clusters and node distribution and can be misleading. However, when considering all metrics, characteristic metric behavior can be observed that reveals information about the clustering's node distribution, and amount of clusters.

To demonstrate this effect, we consider the two most extreme cases regarding node distribution and amount of clusters. These occur either when there is only one cluster containing all nodes (one-cluster clustering), and when every cluster contains only one node (one-node clustering). Since both types of clustering do not provide any information regarding the hierarchical structure of the design, the expected result should be 0 for all metrics. We evaluate these characteristics on a synthetic netlist example, which contains 1000 gates, split into five partitions, each consisting of 200 nodes, as well as on two real designs: the `aes` design for one-cluster clustering, and the `present` design for one-node clustering.

As can be seen in Figure 3, both extreme clustering cases have specific effects on the metric behavior: F1 overestimates the quality of one-node clustering, while NMI overestimates one-cluster clustering. Although the metric values are not high, they can suggest a better quality of clustering than is actually achieved.

In general, when considering metric behavior, we identified the following correlations:

- **Node Distribution**: If either AMI, NMI, and ARI are high, but F1 and nF1 are significantly lower, or if AMI, NMI, and F1 are high and ARI and nF1 are significantly lower, then the clustering contains a similar amount of clusters as the partition. In each case, some scores are low, because the number of nodes in each cluster differs significantly between partition and clustering. This commonly occurs when one cluster in the clustering is significantly larger than all other clusters. The clustering then has a similar number of clusters, but the nodes are distributed too evenly over all clusters, with too few in the largest, and too many in every other cluster. The clustering may seem like a good convergence to the partition, but it does not reveal useful information about the design. The more distinct the peaks and minima in the score behavior, the more extreme is the mismatch in node distribution.

- **Split Partitions**: If more clusters are found than exist in the ground truth, the cluster can be improved manually through merging smaller clusters together and bring them to the size of the corresponding ground truth cluster. However, this is only possible if there exists previous knowledge regarding the size and number of possible partitions. This can also be seen in the score behavior: two scores, AMI and NMI are high, and NMI $>$ ARI $>$ F1 $>$ nF1. Such behavior indicates that there are clusterings suitable for a merge. Otherwise, the node distribution, like in the previously discussed case, is unsuitable for a merge because the basic node distribution does not match the partitioning.

To ensure the clustering matches the partitioning in view of cluster number, node distribution, and node assignment, an equally high value for all scores is crucial. Minor variations ($\sim< 0.15$) between nearby scores can be neglected. Larger differences must be analyzed, as this points to a characteristic of the clustering that does not match the characteristic of the ground truth. New algorithms and methods must be tested using all metrics on order to understand their behavior and judge their performance.

5.5 Evaluation of Cluster Quality

Figure 4 shows the cluster quality and metric behavior for each of the four clustering methods. The best result was selected for methods that require a parameter. We assume that no knowledge regarding the design, other than the design size is known. Other approaches assume some knowledge regarding the design, for example cluster number and relative size. This would also significantly impact our results. In particular, results that show a behavior of Split Partitions will greatly improve, as this knowledge allows for the merging of these clusters. However, for the sake of a fair comparison between our methods, and to demonstrate the flexibility of our methods, we evaluate only this knowledge-free use-case.

In general, our graph-based methods are unable to achieve useful results for small designs. Both the `i2c` and the `present` designs show very low scores for all metrics. We hypothesize that the small number of logic gates do not provide a significant enough sample to leverage the strengths of structural analysis.

Considering the other two cryptographic algorithms, the `BLAKE` and the `AES` design, the results differ significantly. The `AES` displays nearly perfect results for all clustering algorithms, except for *MCL*. The behavior of the *MCL* results indicate that while the clustering is good, too many clusters were created compared to the ground truth, i.e. a clear case of Split Partitions. The `BLAKE` design, which is a hash-based cryptographic function, appears to be more difficult to cluster. Only *Leiden-CPM* provides adequate results for all metrics. This may be due to the strongly interconnected nature of hash functions.

The `altor` and `aquarius` CPUs both show good scores for *Leiden-CPM*, adequate scores for *Louvain* and *Leiden-Mod*, and insufficient scores for *MCL*. Except for *Leiden-CPM*, the other methods suffer from an incorrect Node Distribution, where the number of clusters and partitions are similar, but the distribution of nodes between them is not ideal.

The `ethmac` is the clearest demonstration of the necessity for multiple metrics. Only *Leiden-CPM* is able to provide a good score across several metrics. All other clustering methods show a Node Distribution effect, as they are unable to handle the large size discrepancy between the first cluster and subsequent clusters (see Figure 1). However, when considering only the NMI, this effect would not be discovered.

The `FPU` provides one of the best results, except for the *Leiden-Mod* algorithm, which suffers from Split Partitions. It is notable, that with increase design size, *MCL* seems to provide a better result. A visualization of the resulting clusters and the ground truth is shown in Figure 5, for *MCL*. Visual inspection shows a nearly perfect recovery of the original partitions, with minor errors within the most interconnected parts in the center of the design.

For the `aquarius-mem`, the *Leiden-CPM* again significantly outperforms all other methods. However, when considering the node distribution, the uselessness of this result becomes clear. This design contains a large memory block, which overshadows every other functional submodule in the design. This is also reflected in the metric behavior for the *Leiden-CPM* algorithm, as the F1 score shows that the clustering does not correctly map the gates to their partitions.

In general, the *Leiden-CPM* algorithm often provides the best result, independent of design

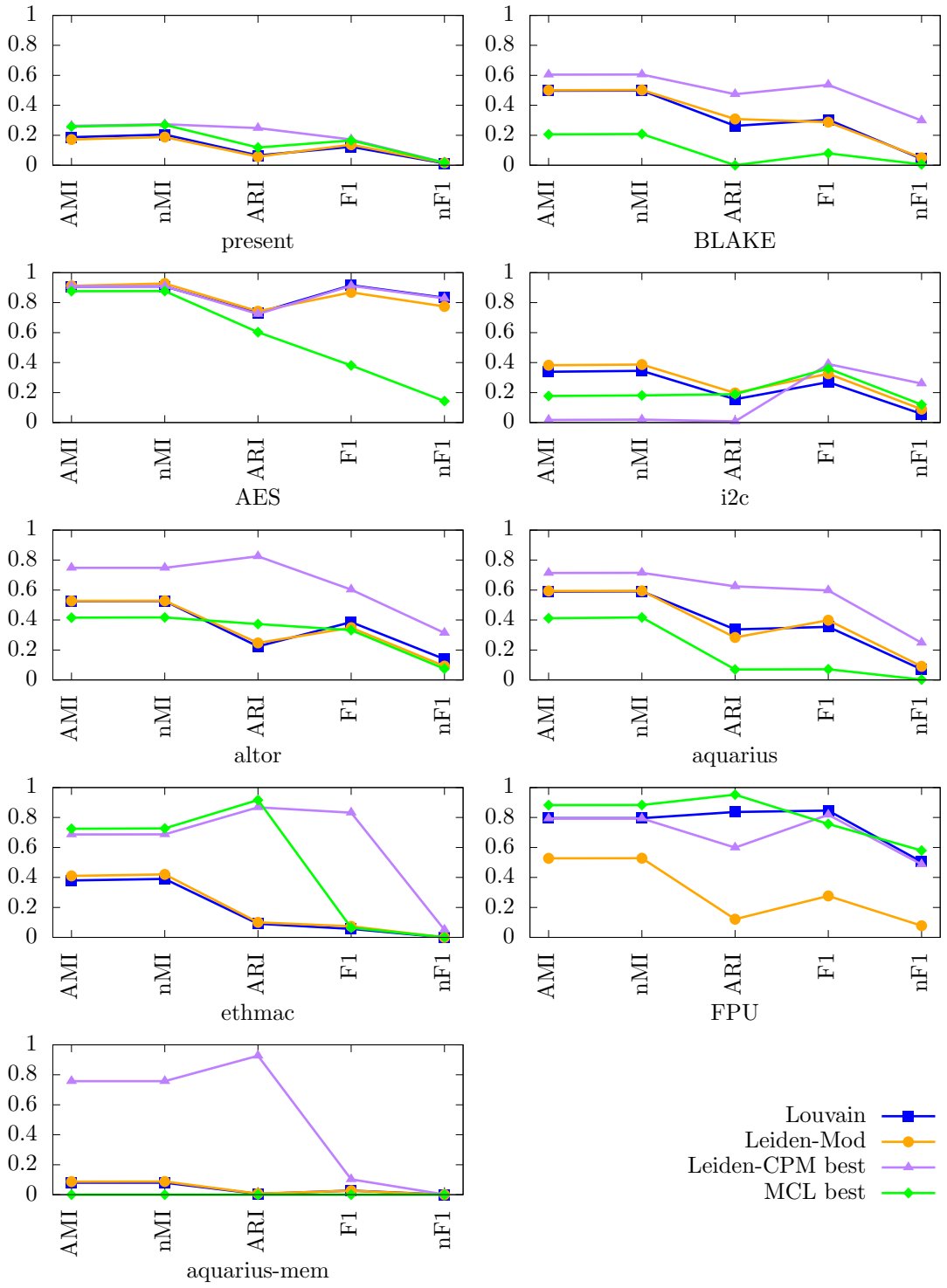
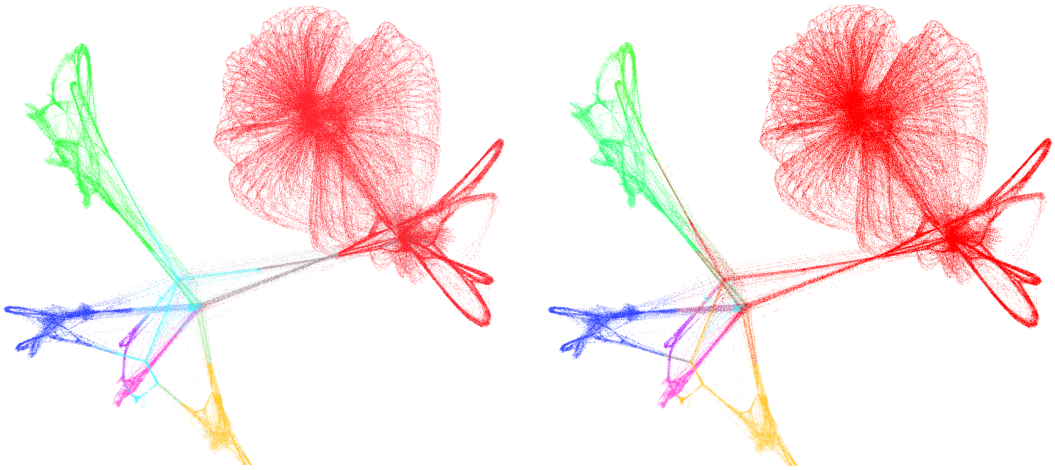


Figure 4: Results per Design for each Metric

size or type. The method is unaffected by uneven Node Distribution. *Louvain* and *Leiden-Mod* perform very similarly, although *Louvain* seems to provide a better result for larger designs. *MCL* is particularly strongly affected by uneven Node Distribution, and does not show its true capabilities in smaller designs.



(a) Partition for `FPU` (b) Clustering for `FPU`

Figure 5: Visual comparison of the `FPU` design

5.6 Performance Analysis

As hardware design netlist grow in gate count and complexity, it is crucial that the RE algorithms can scale at the same rate. Netlist partitioning is the very step that needs to handle large design sizes, as only later steps can profit from the divide-and-conquer benefits.

In this section, the graph clustering approaches are analyzed for their performance with increasing netlist size (i.e. the number of nodes $\left|\bar{V}_0^0\right|$ in the flat netlist $\bar{G}_0^0$, abbreviated as $|V|$) and netlist complexity (i.e. the number of edges $\left|\bar{E}_0^0\right|$, abbreviated as $|E|$). To this end, the computational complexity is presented based on the algorithm descriptions. Based on the experiments described previously, it is also possible to evaluate the real-world-performance, run-time results are summarized in Table 1.

Due to the randomized behavior of the *Louvain* algorithm, the computational complexity cannot be given in a closed form. As the algorithm uses a low number of repeats to compute a quality function for all edges incident to all nodes, the complexity can be estimated as $O(k*|E|)$, in which k is the number of repeats of the calculation to achieve the local minimum of the modularity. If k does not scale with $|V|$ or $|E|$ (as argued in [10]), the result is a linear complexity over $|E|$ or a linear complexity over $|V|$ if the netlist graph is sparse. Run-time analysis presents similar results, with a strong correlation between design size and run-time. For designs with a large disparity between the size of the largest cluster and other clusters, including the `BLAKE`, `ethmac` and `aquarius-mem`, *Louvain* not only provides subpar results, but also requires long run-times. When considering the underlying algorithm, where two clusters are always grouped together to achieve maximum modularity, it becomes clear that such designs require numerous repetitions to find a local maximum.

Similar to the *Louvain* algorithm, the computational complexity of the *Leiden-Mod* algorithm cannot be presented in closed form. It also uses a low number of repeats to compute a quality function for all edges incident to nodes in a working set. However, as the algorithm does not consider each node for every repetition, the working set decreases asymptotically, which leads to a lower computational complexity compared to *Louvain* [11]. Again, this is supported by run-time analysis, where, while the correlation between gate number and run-time persists, *Leiden-Mod* runs significantly faster than *Louvain* for all designs. Furthermore, the algorithm is, from a run-time point of view, largely unaffected by uneven Node Distributions.

As the quality functions modularity and CPM do only differ in constant elements (see A4 and A5 in [11], the use of the CPM quality functions does not increase or decrease the computational complexity of *Leiden-CPM* compared to the modularity-based *Leiden-Mod* algorithm. In fact, the run-times for both methods are very similar, with *Leiden-CPM* only showing an increase in run-time for the `aquarius-mem` designs. Since *Leiden-CPM* is also able to cluster the single large cluster correctly, this is an acceptable trade-off for a better quality result.

The computational complexity of the *MCL* algorithm is given in [12] as bounded by the average of the node degree. As $\sum_{v \in V}(\deg(v))/|V| = 2 * |E|/|V| = O(1)$ it follows that the complexity of this algorithm is $O(|E|)$. Thus, *MCL* scales linearly with the netlist complexity. For typical netlists, which have a sparse netlist graph, this means that *MCL* scales roughly linear with the number of nodes in the design. Table 1 supports this analysis, with a strong correlation between run-time and number of gates. However, *MCL* seems more strongly affected by design size than the other algorithms, such that the computation did not complete within reasonable time ($< 3h$) for the largest design.

In conclusion, the algorithms used in this paper tend to have linear computational complexity over netlist size and netlist complexity. *Leiden-Mod* and *Leiden-CPM* compute clusters in a matter of seconds, or few minutes for large designs, while *Louvain* and *MCL* both require significantly longer run-times for large designs.

5.7 Qualitative Comparison of Partitioning Methods

When comparing different partitioning methods, we not only compare the quality of the results, using the metrics described in section 4. We also consider the scalability (inverse to the computational complexity), as this has a significant effect on the usability of the method in real-world netlist RE. Furthermore, depending on the attack scenario, more or less knowledge regarding the design will be available for the RE process. It must be noted that if more knowledge is required for a method, this does not necessarily make it less applicable to certain scenarios, as there are many cases where this information will be available. However, considering the amount of knowledge needed again allows for a consideration of the applicability to real uses cases. Finally, we also evaluate whether the method requires parameter choices. Many parameters signify that a method may not easily be transferable to different design types and sizes, and is thus less flexible.

We consider a number of data-path and graph based partitioning methods (see Table 2). We describe the type of method, the scalability, how "knowledge-free" the method is, the flexibility of the method due to the required parameters, and finally, the quality of the results. In general, a high score in each of these categories is desirable.

One of the earliest works leveraging data-path information proposes a partitioning approach based on finding data-words and their connection, and using these to partition the design. These words are found by using similarity scores; in this case the feasible cut and shape of each boolean gate. Gates with similar structure and functionality are then grouped into data-words.

Connections between these words are established by propagating signals from one word to the next. Finally, submodules between all possible combinations of data-words are carved out, and compared against a golden model in order to gain functional equivalence [3], [4]. While this approach provides good results for smaller designs, the overhead is very large, and without assumptions made regarding the size of data-words, reverse engineering real sized designs is impossible. Especially in larger designs, the number of candidate words becomes difficult to handle, and needs to be significantly reduced by hand using information about the design that is not known in a real life scenario. Furthermore, even from a few candidate words, many candidate submodules may be identified, which then need to be checked against the golden model library. Thus, the effort to remove bad submodules and to ensure good coverage is very high. The largest analyzed design consisted of $\approx 300k$ gates, which required several hours of analysis. For this analysis, assumptions regarding word size were made, so that the design could be analyzed at all. However, only the depth and size of the similarity scores are used as parameters, and thus the flexibility is not strongly affected. No metric is used to evaluate the quality of the results, however, the coverage of the extracted design is reasonable.

A later work [5] concludes that the above discussed approach is the best for larger designs, but proposes a control signal based partitioning method for small designs, and a Principal Component Analysis (PCA)-based method for design where information regarding the number of words is known or can be guessed. This third method is parameter dependent, while the control signal based method requires the identification of control signals within the design. All three methods again depend on identifying data-words within the design, and carving out candidate submodules between these words. The paper does not discuss the difficulty of extracting the correct combination of words, such that the candidate submodule actually consists of the desired functional submodule. Instead, the resulting data words are compared to the data words extracted from RTL code, and compared using a NMI score. Finally, the paper concludes that more accurate methods are necessary, as all methods are unable to achieve a good results on a data-word level. As results are not calculated on a gate-level, but only on a data-word level, a fair comparison between data-path based methods and graph based methods is difficult.

The earliest work on graph-based partitioning for RE uses the ncut algorithm to partition a large design into several smaller clusters [6]. No metric is used to evaluate the quality of these clusters, choosing instead to present only the assignment of gates to each cluster and partition. Furthermore, the single design evaluated is small, so it is unclear how well this method scales. However, the method is significantly faster than most of the data-path based methods, and provides similar results. Furthermore, as the ncut algorithm is unsupervised, no knowledge regarding the design, and no parameters are required, making this method extremely flexible.

In 2018, [7] also used a graph-based approach - the Louvain algorithm - and incorporated layout information as edge weights to improve the results. These are evaluated using the accuracy of the assignment of gates to a cluster, and several designs, including some larger designs, are evaluated. While no experimental run time analysis was done, the computational complexity of the algorithm is discussed. As we also evaluate this method, if without layout information, we will not compare against this work.

A graph-based approach requiring more input information was presented by [8] in 2019. Here, it is assumed that a block diagram of the design, together with the relative size of each block is known. The method then matches each node in the graph to one of these blocks, by calculating a geometric embedding for each node, and clustering based on these embeddings. This requires a number of parameters, and while not comparable to other graph-based methods in terms of scalability, the quality of the results are similar.

Recently, the use of graph neural networks (GNNs) for RE have become more common. Since

Table 2: Qualitative Comparison of Partitioning Methods

Method	Type	Scalability	Knowledge-free	Flexibility	Result Quality
WordRev [3]	data-path	low	medium	medium	medium
control signal [5]	data-path	high	high	high	medium
PCA [5]	data-path	low	medium	medium	medium
n-cut [6]	graph	high	very high	high	low
Embedding [8]	graph	low	medium	medium	medium
GNNs [9]	graph	very low	medium	low	very high
Louvain	graph	high	very high	high	medium
Leiden-Mod	graph	high	very high	high	low
Leiden-CPM	graph	high	very high	medium	high
MCL	graph	high	very high	medium	medium

a netlist can be easily converted to graph form, and with recent works, which make GNNs more scalable, and thus usable for larger graphs, GNNs seem a promising solution to many of the REs problems. [9] proposes the use of GNNs for both netlist partitioning and identification. The data-set is focused on algorithmic functions, and while run-times are very high compared to other methods, the results surpass those previously presented. Main weaknesses include the dependency on the correct training data, and the large number of hyper-parameters required for the GNNs model design.

The methods presented in this paper are all highly scalable, with results for between a few seconds and a few minutes even for very large designs. Especially *Leiden-Mod* and *Leiden-CPM* are able to handle huge netlists easily, and thus also allow for iterative processing of such designs. All four methods require minimal knowledge regarding the structure of the design, with some information regarding the size of the design being required to pick good parameters for *Leiden-CPM* and *MCL*. The flexibility of these two methods is a only medium, as parameters are required. However, both *Leiden-Mod Louvain* require no parameters, resulting in a high flexibility. The best results, based on the here proposed metrics, with the above analyzed benchmarks, are produced by *Leiden-CPM*; *Louvain* and *Leiden-Mod* also produce useable results.

As can be seen in Table 2, many of the compared methods excel in one or more category, however, graph-based methods not only provide the best quality of results, but also achieve good scalability, require little knowledge about the design, and either require no parameters, or allow for easy parameter choice, making them usable for many different types of designs.

6 Conclusion

In this work, we evaluated four graph-based partitioning methods for netlist reverse engineering, including an analysis of the quality of results, computational complexity and run-time, for real-life designs. We also introduced a number of metrics to fairly and efficiently measure the quality of results, and conclude that the evaluation using only a single metric is insufficient to truly judge the capabilities of new partitioning methods. We compare our methods to other graph-based and data-path based partitioning methods, and surmise that graph-based methods provide the best trade-off of scalability, flexibility and partition quality.

References

[1] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Models, Methods, and Metrics," *Proc. IEEE*, 2014.

[2] M. Xue, C. Gu, W. Liu, S. Yu, and M. O'Neill, "Ten years of hardware Trojans: a survey from the attacker's perspective," English, *IET Comput.s & Digital Techniques*, 6 2020.

[3] W. Li *et al.*, "WordRev: Finding word-level structures in a sea of bit-level gates," in *Hardware-Oriented Secur. Trust (HOST), 2013 IEEE Int. Symp.*, 2013.

[4] P. Subramanyan *et al.*, "Reverse Engineering Digital Circuits Using Structural and Functional Analyses," in *Emerg. Topics Comput., IEEE Trans.*, 2014.

[5] T. Meade, K. Shamsi, T. Le, J. Di, S. Zhang, and Y. Jin, "The Old Frontier of Reverse Engineering: Netlist Partitioning," *J. Hardware Syst. Secur.*, 2018.

[6] J. Couch, E. Reilly, M. Schuyler, and B. Barrett, "Functional block identification in circuit design recovery," in *2016 IEEE Int. Symp. Hardware Oriented Secur. Trust (HOST)*, 2016.

[7] M. Werner, B. Lippmann, J. Baehr, and H. Gräb, "Reverse Engineering of Cryptographic Cores by Structural Interpretation Through Graph Analysis," in *2018 IEEE 3rd Int. Verification Secur. Workshop (IVSW)*, 2018.

[8] B. Cakir and S. Malik, "Revealing Cluster Hierarchy in Gate-level ICs Using Block Diagrams and Cluster Estimates of Circuit Embeddings," *ACM Trans. Des. Autom. Electron. Syst.*, 2019.

[9] L. Alrahis *et al.*, "GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 2021.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Statistical Mechanics: Theory Experiment*, 2008.

[11] V. A. Traag, L. Waltman, and N. J. van Eck, "From Louvain to Leiden: guaranteeing well-connected communities," *Sci Rep*, 2019.

[12] S. M. van Dongen. "Graph Clustering by Flow Simulation." (2000), https://dspace.library.uu.nl/bitstream/handle/1874/848/full.pdf?sequence=1&isAllowed=y.

[13] J. Baehr, A. Hepp, M. Brunner, M. Malenko, and G. Sigl, "Open Source Hardware Design and Hardware Reverse Engineering: A Security Analysis," en, in *Euromicro Conf. Digital System Des.*, 2022.

[14] G. Rossetti, L. Pappalardo, and S. Rinzivillo, "A novel approach to evaluate community detection algorithms on ground truth," 2016.

[15] G. Rossetti, "RDyn: Graph benchmark handling community dynamics," *J. Complex Networks*, 2017.

[16] OpenCores.org. "OpenCores Homepage." (2021), https://opencores.org/.

[17] J. Strömbergson, *secworks*, 2018. https://github.com/secworks?tab=repositories.

[18] M. Ludwig, A. Hepp, M. Brunner, and J. Baehr, "CRESS: Framework for Vulnerability Assessment of Attack Scenarios in Hardware Reverse Engineering," in *2021 IEEE Physical Assurance Inspection Electron.s (PAINE)*, 2021.

[19] J. Baehr, A. Bernardini, G. Sigl, and U. Schlichtmann, "Machine learning and structural characteristics for reverse engineering," *Integration, VLSI J.*, 2020.