Meta-Scanner: Detecting Fault Attacks via Scanning FPGA Designs Metadata

Hassan Nassar*, Jonas Krautter, Lars Bauer, Dennis Gnad, Mehdi Tahoori* and Jörg Henkel*
* *Karlsruhe Institute of Technology (KIT)*
* {hassan.nassar, mehdi.tahoori, henkel}@kit.edu

***Abstract*—With the rise of big data, processing in the cloud has become more significant. One method of accelerating applications in the cloud is to use FPGAs to provide the needed acceleration for user-specific applications. Multi-tenant FPGAs are a solution to increase efficiency. In this case, multiple cloud users upload their accelerator designs to the same FPGA fabric to use them in the cloud. However, multi-tenant FPGAs are vulnerable to low-level denial-of-service attacks that induce excessive voltage drops using legitimate configurations. Through such attacks, the availability of cloud resources to non-malicious tenants can be hugely impacted, leading to downtime and thus financial losses to the cloud service provider.**

In this paper, we propose a tool for offline classification to identify which FPGA designs can be malicious during operation by analyzing the metadata of the bitstream generation step. We generate and test 475 FPGA designs that include 38% malicious designs. We identify and extract five relevant features out of the metadata provided from the bitstream generation step. Using 10-fold cross-validation to train a random forest classifier, we achieve an average accuracy of 97.9%. This significantly surpasses the conservative comparison with state-of-the-art approaches, which stands at 84.0%, as our approach detects stealthy attacks undetectable by existing methods.

I. Introduction

Field Programmable Gate Arrays (FPGAs) are now heavily utilized as versatile accelerators in the cloud computing domain [1], [2], [3], [4], where users can realize almost arbitrary circuits on these programmable logic chips. The ever-increasing amount of programmable resources per FPGA chip enables fine-grained virtualization to optimize efficiency and utilization [5]. Virtualization and multi-tenancy (multiple users, i.e., tenants share the resources of the same FPGA) is heavily discussed in the literature [6], [7], [8]. However, researchers demonstrated unsolved security issues of FPGA multi-tenancy in the form of remote fault attacks [9], [10], [11], [12]. The attacks have been escalated to actual cloud devices in the Amazon AWS instances [13], enabling large-scale Denial-of-Service attacks that can result in financial loss for the Cloud Service Provider (CSP). The attacker causes strong fluctuations in the FPGA's Power Distribution Network (PDN), resulting in its sudden shutdown. The attacker achieves this by implementing several thousands of oscillators on the FPGA [12].

This work was partially funded by the "Helmholtz Pilot Program for Core Informatics (kikit)" at Karlsruhe Institute of Technology and by the German Federal Ministry of Education and Research (BMBF) through grant 01IS23066 as part of the Software Campus Project "HE-Trust".

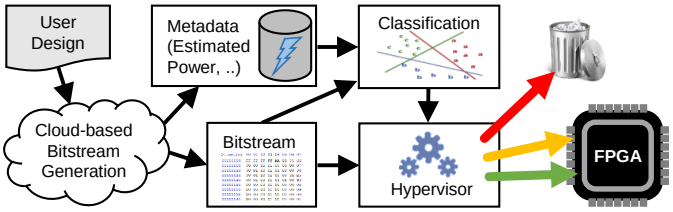


Fig. 1: Basic principle of our proposed Meta-Scanner and loading flow. Bitstreams classified as being high-risk are not loaded. Other bitstreams are loaded but with careful placement.

To address these security issues, offline and online countermeasures have been proposed [14], [15], [16], [17], [10], and basic design rule checks are already employed by the industry [13]. Existing offline countermeasures based on bitstream checking [14], [15], [16] fail to identify the most recent malicious designs. For instance, seemingly benign circuit designs, based on minor modifications to the AES encryption modules, have been demonstrated as capable of inducing timing faults or causing a Denial-of-Service [18]. These seemingly-benign circuits achieve strong PDN fluctuation through specific input patterns instead of using simple oscillators. Moreover, it is also possible to build an attack by using multiple malicious tenants in a coordinated way, even though none alone would lead to a successful attack [19]. Thus, attacks are getting more stealthy and are harder to detect.

Two online methods to disable malicious tenant designs during operation have been proposed [17], [10], but both have restrictions on the type of malicious designs they can prevent and how fast they can do that. Thus, to effectively stop an attack, a hypervisor must know upfront which tenant could be potentially malicious, as its adversary effects can already become effective a few microseconds after it was deployed. If not, targeting the malicious tenant would take several milliseconds and the attack will be successful [10].

In this work we propose a machine-learning-based classification that can be used offline on tenant designs before loading them to an FPGA. We show the basic flow of our approach in Fig. 1. In the cloud, a tenant design is compiled into a bitstream with accompanying metadata, such as estimated power consumption. Our classification scans the metadata as input and categorizes the tenant's designs into three categories. These categories correspond to its risk level of becoming a potential threat to the integrity of other tenants or the entire system. Based on the scanner, hypervisors can choose a suitable mapping of tenant designs to different FPGAs that can maximize security through additional online countermeasures

as in [17], [10].

Tenant designs of the high-risk category (RED) are banned from being loaded to any FPGA region. Designs of the low-risk category (GREEN) can be arbitrarily placed together with other low-risk tenant designs on the same FPGA. The hypervisor would deploy at most a single mid-risk (YELLOW) tenant design on te same FPGA. In this case, existing runtime countermeasures [17], [10] in case of detected malicious activity will be able to disable the YELLOW tenant design. If we deploy more than one YELLOW tenant design to the same FPGA, the online tools cannot stop both of them fast enough, in case of an attack.

Altogether, **our novel contributions** are as follows:

- We propose an offline FPGA design classification in which we identify and extract five relevant features from a tenant design, using metadata from the bitstream generation step, to categorize its risk level.
- Our proposed classifier covers more types of malicious designs than any state-of-the-art solution. It reaches an average cross-validated accuracy of 97.9%, whereas state-of-the-art checkers only achieve accuracies up to 84.0% in a conservative comparison.
- We generate a comprehensive set of 475 tenant designs based on malicious and benign logic.[1] We label them using three risk classes RED, YELLOW, and GREEN.

The organization of the rest of this paper is as follows. We describe the necessary background and state-of-the-art approaches in Section II. Our main contributions, the offline design classification and the required metadata scanning, are explained in Section III. Section IV presents the generation of the bitstreams. We present our results and analysis in Section V. In Section VI, we discuss the limitations and advantages of our work, and Section VII provides conclusions.

II. Background and Related Work

A major aspect of this work is to focus on the remaining blind spots of existing countermeasures against fault attacks in cloud FPGAs. To get sufficient background, we first explain the current assumptions on multi-tenant FPGAs. Then we detail the existing attacks and their consequences to Cloud Service Providers (CSPs). Moreover, we elaborate on which attacks can be performed in the cloud and which further countermeasures are available.

A. Multi-tenant FPGAs for Cloud Applications

Multi-tenant FPGAs are a heavily discussed topic. It has interest from both academia [8] as well as industry, e.g., IBM [20]. The idea stems from the fact that FPGA resources increase and one user, i.e., tenant, might not need to use all the resources on the FPGA. Hence, to increase efficiency, the FPGA can be shared by different tenants. To manage the tenant designs, a static part of the FPGA is used by the CSP to manage the communication and interfaces of the different tenants. The tenant designs reside in a dynamic part, with several accelerator slots that can be used by tenants [5].

[1] https://gitlab.kit.edu/hassan.nassar/Meta-Scanner

The tenant design focuses mainly on the application accelerated by the user [8], [5], [21], [20]. Any memory controllers or PCIe subsystems would not reside in the tenant region. Such components would rather belong to the static design controlled by the CSP to avoid conflicts between the tenants when using shared resources such as off-chip RAM. AWS already does this in its commercial single-tenant systems [22].

B. Cloud FPGA Attacks

The interest in multi-tenant FPGAs sparked security concerns [6], [7], [23]. As in cloud FPGAs, physical attacks that do not require physical access to the chip become increasingly concerning [12], [24]. In the literature, passive side-channel [24] and active fault attacks [12], [9], [25], [26] are mentioned for cloud FPGAs. This work solely focuses on the latter.

In such fault attacks, high power-consuming designs cause instability in the power distribution network. When the attacker uses a large enough design, the whole FPGA or its power supply can crash. This requires manual power cycling to recover the system [12], [25], leading to a major loss of availability. Recently, it was shown that Denial-of-Service attacks work in commercial FPGA clouds, with only minimal modifications [13]. The authors also show that significant financial loss can be expected for the CSPs, when Denial-of-Service attacks are performed, leading to longer downtimes of the FPGA infrastructure.

The initial versions [12], [9] of FPGA fault attacks used ring oscillators or other combinational loops for high power consumption. However, these clearly malicious circuits can also be replaced by more stealthy variants, with the first step being synchronous flip-flops [25]. Later, it was shown that intermittent short-circuits could be caused by certain Block RAM access patterns, causing sufficient voltage drop and even bitflips in configuration memory [27]. Another alternative is 'glitch amplification', which uses a fast-clocked flip-flop with a large output network designed to have many glitches and thus high power consumption [28]. A wider overview of similar circuits optimized for high power consumption is presented in [15].

All of these mentioned circuits use uncommon circuit structures. However, it has also been shown that combining multiple benign synchronous IP modules, e.g., AES, can be used for attacks [18], [29]. Moreover, the attack can be distributed to multiple malicious tenants launching a coordinated attack [19].

C. Offline Countermeasures against Fault Attacks on Cloud FPGAs

From the malicious designs in Section II-B, only some combinational loops are detected by the FPGA CSPs through typical design-rule checks (DRC) that are not necessarily meant for security. In the literature, more sophisticated checks have thus been proposed in [14], [15], [16]. Reverse-engineering is used by [14], [15] to perform security checks. They look into detecting patterns to find malicious elements. Ref. [14] presents a heuristic to check for high-fanout-nets that are often used in attacker designs to toggle large amounts of

logic synchronously. Regarding the method presented in [15], the authors use the reverse-engineered bitstream to recreate the netlist. From the netlist, they can find any self-oscillating structures that might escape the DRC done by the CSPs. Moreover, [11] offers a similar approach to ours using ML on bitstreams. They are limited to work on full bitstreams, lacking support for partial bitstreams, and they focus on detecting self-oscillating structures. They improve over previous works by detecting hidden malicious designs within benign designs. Similar to [11], [16] provides initial results on training a Convolutional Neural Network (CNN) to detect self-oscillating structures.

However, all these offline countermeasures cannot detect recent malicious designs. As even standard IP-core modules such as AES and shift registers can be used to provoke crashes [18], [29] because they seem benign. Very recently, [30], [31] showed initial results for detecting cryptographical-circuits-based malicious designs. However, they cannot detect many sequential malicious designs, such as shift registers [18] or RAM-based malicious designs [27] which escape detection by all the state-of-the-art solutions. We compare our approach with all the tools from the state-of-the-art in Section V.

D. Online Countermeasures against Fault Attacks on Cloud FPGAs

Another mitigation approach is to detect attacks online, and try to prevent them, i.e., ways for *detection* and for *prevention*. For the detection of attacks, a delay line can be used to detect voltage drops [12]. By distributing multiple of them, the exact location of the attacker can be found in about 9.9-21.0 μs [17], but some attacks succeed faster than that [10].

Preventing attacks can be more challenging, as FPGAs are not designed to disable an entire region rapidly. When the attack relies on an external clock, a clock disable will be sufficient to stop the attack quickly enough [12], [17], [13], but it cannot prevent attacks with a self-generated clock [10]. To prevent such attacks, *LoopBreaker* can stop attacks at runtime by quickly reconfiguring all interconnects of the malicious tenant to high impedance in about $1.5\mu s$ [10]. However, due to limitations in the reconfiguration time, LoopBreaker needs to know in advance which tenant shall be stopped before the attack even starts. LoopBreaker can quickly stop an ongoing attack if and only if that information is available upfront.

E. Summary of Countermeasures

We analyzed all the existing countermeasures and can summarize their shortcomings as follows:

- Existing FPGA design scanning variants need to reverse engineer and analyze on the netlist level deeply, but they still can only detect designs with **clearly malicious constructs** [14], [15], [16].
- Existing runtime approaches can detect the malicious region [17], but they might be too slow or have limitations that only one malicious region can be disabled fast [10].

Because of these shortcomings, it is possible that seemingly benign designs that escape detection by state-of-the-art can perform successful attacks. Thus, we take a different approach to FPGA design checking in this paper. Our approach does not require reverse engineering or a deep analysis on the netlist level. Nevertheless, our approach can still rate the risk potential of an FPGA design in using its metadata to detect complex attacks to allow the runtime approaches to work more effectively.

III. Meta-Scanner: Identifying Malicious FPGA Designs

Our main goal is to develop an offline scanner that allows the CSP to analyze tenant designs before uploading them to an FPGA. This should be done without a time-consuming and extensive netlist analysis, and at the same time, it should be sufficient to complement and assist existing runtime countermeasures [10], [17]. We classify tenant designs into three categories: high risk (RED), mid risk (YELLOW), and low risk (GREEN), which removes the burden from existing runtime countermeasures to identify the malicious tenant before starting the countermeasure. Our chosen random forest classifier consists of several decision trees. Each decision tree is actually very similar to a simple rule based inference. The main difference is in finding appropriate thresholds for the rules. The ML part can be seen as an automated way to determine the individual decisions and finding the thresholds during training. This ML training step helps to ensure that the rules are not mistakenly overfitted to the known attacks used for training, but that they remain generic enough to also cover other attacks. Additionally, it helps to adjusting to novel attack types as soon as they occur, as retraining is an automatic operation.

A. Threat Model and Assumptions

The threat model We target is a cloud scenario with multi-tenant FPGAs, i.e., multiple tenants share an FPGA in a cloud system with potentially multiple FPGAs. The attacker might rely on *intra-FPGA* coordination, i.e., using multiple regions on a single FPGA *together* to crash the FPGA (see Section II-B). Our focus is mainly on detecting malicious tenant designs. By correctly classifying the risk level of each tenant design, we provide CSPs with the ability to decide whether or not to upload it. We assume that CSPs perform security checks or attestation of the FPGA design through a hypervisor as explained by previous works [32]. Moreover, CSPs can combine our risk classification with other data they might have. Usually, CSPs can have access to more information about their users, e.g., their history of previous tenancy on FPGAs. Hence, they may have some trust metric for the users, which is beyond the scope of our work.

The steps to use our solution are shown in Fig. 1. Normally, a tenant would upload a design as an HDL code or as a netlist to the CSP. The CSP then generates the bitstream and extracts the features (see Section III-C) used by our scanner from the metadata. Then, based on our scanner, the CSP can correctly evaluate the risk category of the bitstream.

The hypervisor should never upload RED tenants (see Fig. 1), as they will very likely exhibit malicious behavior, whereas GREEN tenants can always be uploaded, as they

are incapable of displaying malicious behavior. YELLOW tenants can be uploaded to an FPGA, but special care must be taken, as explained in Section I. When ensuring that at most one YELLOW tenant is executing on an FPGA, then online countermeasures like [10], [17] can aim at the potentially malicious tenant, which allows them to shut it down as soon as it measures any malicious activity. Instead, if two or more YELLOW tenants were on the same FPGA, it would no longer be known which of them started the malicious activity. Thus, the online countermeasures would no longer be able to localize and stop the activity fast enough before a crash occurs.

B. Tenant Design Analysis

To classify tenants accordingly, we start by thoroughly analyzing both malicious- and benign designs (generation of the dataset of tenant designs is described in Section IV), to get an idea of which features would be helpful to detect malicious designs. As typical malicious designs aim at triggering a voltage drop to cause Denial-of-Service (see Section II-B), the most straightforward idea is to use the estimated power consumption of a tenant. However, our analysis shows that this power estimation is very inaccurate for the earlier published malicious designs [28] that used highly regular structures (e.g., mux-based, latch-based, or glitch amplification-based; see Section II-B). The power estimation alone will not be enough to classify the malicious designs properly. However, it is noticeable that the earlier published malicious designs have a highly regular structure and repetitive elements in their design, as they are composed of many relatively small building blocks. We show in Section III-C how to extract and exploit this property of the bitstream metadata for our classification.

Repetitive elements in the bitstream are not always an indicator of malicious tenants, because simple benign tenant designs, which have a low power consumption and are mostly empty. Therefore, they will show a high degree of repetition in their bitstreams as the unused resources will have similar configuration data setting them to blank. Hence, these benign tenants can unintentionally appear like malicious tenants to the bitstream classifier. The observable repetition is because most resources in their tenant region are unused. For example, AES uses only very few DSP blocks. We will have to distinguish the repetition due to repeated attack blocks from repetition due to repeated unused blocks in the bitstream classification.

Complex benign designs like a Bitcoin miner or a cluster of different big designs have a high estimated power consumption and a high utilization with a low degree of repetition. It should be easy to distinguish them from malicious designs with highly regular structures. However, malicious designs that are based on benign modules (e.g., the AES-based attacks [18] explained in Section II), also show a high estimated power and a low degree of repetition, which makes them appear similar to complex benign designs.

C. Metadata Extraction

Our idea is to identify the area utilization of a tenant and its internal regularity by extracting corresponding properties directly from its bitstream. Figure 2 shows the structure of Xilinx bitstreams. It has headers and trailers for synchronizing the bitstream upload and the payload. Internally, the main payload of a bitstream consists of so-called *frames*, i.e., the smallest reconfigurable unit in an FPGA (in the low kiB range per frame, depending on the FPGA family). For every reconfigurable region, the synthesis tools for partially reconfigurable designs create a so-called *blank* bitstream (shown in Fig. 2a) that reconfigures the region into an empty state.

A normal design bitstream for the same region can be seen in Fig. 2b. It has the same structure as the blank bitstream. For unused regions, the frame data is identical to the frame data of the blank bitstream. Hence, any frame with data identical to the corresponding frame in the blank bitstream can be seen as empty.

(a) Blank Bitstream (b) Design Bitstream

Fig. 2: Bitstream Structure for blank bitstream and design bitstream. Both bitstreams will have the same static routing, but the design bitstream will have the content of the frames different.

Based on the bitstream structure, we extract 5 features as follows. Note that for the equations, we use the annotation from Table I.

TABLE I: Annotation of the mathematical explanation for the features

Variable	Explanation
$Bitstream_{Len}$	Number of frames per bitstream
$N_{UFrames}$	Number of unique frames
$N_{BFrames}$	Number of blank (empty) frames
$NonBFrames$	Non Blank Frames

- **Repetition:** The number of non-unique frames. If there are for instance 100 frames with identical data, that adds 100 to the Repetition. Nothing is added to the Repetition for a unique frame (i.e., no other frame has the same data). A higher Repetition indicates a higher risk of self-oscillating structures, as they normally consist of many repeated frames.

$$Repetition = Bitstream_{Len} - (N_{UFrames} + N_{BFrames})$$

- **Utilization:** The number of frames different from the frame data at the same position in the blank bitstream. This helps to identify complex designs that use a large degree of their resources.

$$Utilization = Bitstream_{Len} - N_{BFrames}$$

- **Average Frame Frequency (AvgFrameFreq):** We create a histogram of all non-blank frames in the bitstream, i.e., of those frames that are different than the corresponding frame in the blank bitstream. The frequency of the histogram's bins denotes how many frames belong to that bin, i.e., how

many frames have the same data. For the AvgFrameFreq, we calculate the average over the frequencies and divide it by the largest frequency. If the AvgFrameFreq is near one, it indicates a low degree of repetition, while if it is close to zero, it indicates a higher degree of repetition.

$$AvgFrameFreq = \frac{mean(hist(NonB_{Frames}))}{max(hist(NonB_{Frames}))}$$

- **Standard Deviation of the Frame Frequency (StdFrameFreq):** The metric calculates the standard deviation of the frame frequencies and then divides it by the largest frame frequency. This helps identify how much repetition exists. A low deviation means that there is a high degree of repetition and a high deviation means that there is a low degree of repetition.

$$StdFrameFreq = \frac{std(hist(NonB_{Frames}))}{max(hist(NonB_{Frames}))}$$

- **Estimated Power:** This feature estimates the design's power consumption. It is the only feature not directly calculated from the bitstream but is reported by the synthesis tools after the design is placed and routed. Note that for the Amazon Cloud, the CSP has access to this information, as the place and route of a tenant design is performed under the control of Amazon (see also Section VI).

$$EstimatedPower = VivadoPowerEstimation$$

Using these five features covers all the important aspects of high utilization, high power, regular structures, and regular structures hidden with some irregularities, which we need for classifying the designs. Overall, they were effective enough to keep our accuracy, recall, and precision around 97%. Initially, we experimented with 10 features from the metadata, but through experimentation, we found that the 5 we use are enough. The five features excluded are the most repeated frame, the number of occurrences of the most repeated frame, the average value of all the bitstream words, and the standard deviation of all the bitstream words. The final 5 features we use have some overlaps but cover different aspects. $StdFrameFreq$ and repetition are somehow correlated. In case of a non-hidden attack, repetition is very powerful to detect the attack while StdFrameFreq cannot be of the same strength. However, for cases where a malicious design is hidden within a benign design, repetition cannot really be used on its own, the $StdFrameFreq$ is then more accurate. Therefore, both features are needed. We evaluate the features' relevance to our classification problem in Section V-B using the scikit-learn library [33].

D. Proposed Classification

We demonstrate the feasibility of a machine learning approach on the features enlisted in Section III-C by first manually labeling a set of 475 different tenant designs that were tested on a ZCU102 FPGA board (more information in Section IV) according to our three risk classes and evaluating various classifiers on the set. The tenant designs are labeled according to the following principles:

RED (high risk): These tenant designs contain actual attack circuits, which we intentionally designed as malicious using different approaches both from the literature [9], [18], [25], [27], [29]. The hypervisor should never load them to tenant regions on the cloud FPGAs.

YELLOW (mid risk): If a circuit contains a lot of resources and may be used in combination with another similar design on the same FPGA to invoke crashes, we label it as a YELLOW design. The hypervisor can permit these designs but requires consideration regarding the mapping into FPGA regions. Note that this definition includes completely benign but resource-intensive as well as intentional malicious designs. For instance, additional logic may be added to confuse offline bitstream checker and *hide* the attack, or attackers might use reduced variants of the RED designs based on multiple seemingly-benign IP modules. Multiple YELLOW-labeled tenants should not be present at any given time in the FPGA to prevent attacks. If at most a single YELLOW design is deployed per FPGA, existing runtime countermeasures [17], [10] will be fast enough to disable it in case of any detected malicious activity (see Section III-A).

GREEN (low risk): Tenant designs from the GREEN category are considered harmless and can be arbitrarily placed into different FPGA regions by the hypervisor. They are neither resource-intensive nor contain known malicious structures such as self-oscillating circuits. Attacks are highly unlikely even if combined with YELLOW designs on the same FPGA.

To correctly classify the tenant designs, we use the insights from the bitstream analysis in Section III-B to extract the metadata. By performing the metadata extraction based on the template of the empty tenant region, we can use this metadata to train a lightweight classifier that does not need any complex models to reach far superior results compared to the state-of-the-art as we show in Section V-C.

Based on the recommendations in [34], we evaluate 10-fold cross-validation for different classification methods. We tested a support vector machine (SVM), a multi-layer perceptron (MLP), and a random forest classifier. We determined random forest performed the best on our dataset and used it in all further experiments. We use the scikit-learn python library [33] to implement the classifier and focus on optimizing the recall for classification of the RED bitstream class by setting the class weights to 200, 30 and 1 for RED, YELLOW and GREEN respectively. This approach prevents the misclassification of attack bitstreams into a lower-risk class. Thus, it maximizes the security at the cost of very few lower-risk bitstreams not being loaded to the FPGA.

E. Flow of using Meta-Scanner

Our proposed Meta-Scanner is easy to use. CSPs will have to deploy a training phase over the existing tenant designs and known malicious designs, then use the trained Meta-Scanner on any tenant design being uploaded.

1) Training Phase: Algorithm 1 summarizes the steps for the training phase by the CSP. It has first to estimate the floorplanning for its different FPGAs to partition them into several tenant regions. The CSP already has several tenant

designs from its previous users. For each tenant region, generate the blank bitstream to be used as a reference for the feature extraction. Then for each tenant design, the bitstream for all the fitting tenant regions has to be generated, and extract the features from the metadata. If no data about whether the design is malicious or not, it has to be uploaded to an FPGA to get the ground truths. Based on the labeled tenant designs, the classifier has to be trained to be used to scan new tenant designs.

Algorithm 1: CSP Classifier Training

```
Input: List of tenant designs
Output: Trained classifier model
ImplementDifferentFloorplans();
foreach tenant region do
    GenerateBlankBitstreams();
    foreach tenant design do
        GenerateBitstream();
        CompareToBlankBitstream();
        ExtractFeatures();
        UploadToFPGAAndGetGroundTruths();
    end
end
FeedLabeledDataToTrainClassifierModel();
```

2) Scanning Phase: The trained classifier is continuously used to scan new tenant designs to find out whether they are malicious or not and guide the upload of tenant designs on FPGA. The steps of using the scanner are summarized in Alg. 2. The user usually uploads the tenant design as a synthesized netlist [13]. Therefore, an estimation of the resource needed exists. Based on this estimation, the CSP can choose a suitable tenant region from the floorplan. The CSP generates the bitstream for the tenant region and extracts the features from the metadata. The scanner uses the features extracted to get the label for the design. Based on the label, the tenant design is either banned (RED), uploaded with consideration (YELLOW), or uploaded and trusted (GREEN).

IV. Dataset Generation

To evaluate the effectiveness of our scanner in fulfilling its goal, we generated the dataset of the bitstreams. In Table II we summarize the terminology used to describe the dataset generation.

We built a set of bitstreams to extract the metadata and test our solution. The set is based on 26 basic designs, of which nine are malicious and seventeen are benign. We create 475 different tenant designs by configuring, combining, and modifying these 26 basic designs. Six of the nine malicious basic designs are from the state-of-the-art mentioned in Section II-B. Moreover, we implement three new malicious designs, similar to the AES malicious design, that we detail later in Section IV-B. The sixteen benign basic designs are based on the groundhog benchmark [35], ISCAS benchmark [36], Open Cores designs [37], Berkeley benchmarks [38], Xilinx HLS tutorials [39], and RISC-V dual core [40]. In addition to these

Algorithm 2: Tenant Design Classification and FPGA Deployment

```
Input: Tenant design netlist
Output: Label
// Step 1: Synthesize netlist of the
   tenant design
// Step 2: Estimate a fitting tenant
   region
EstimateTenantRegion();
// Step 3: Perform Place and Route
PerformPlaceAndRoute();
// Step 4: Compare to blank bitstream
CompareToBlankBitstream();
// Step 5: Extract features
ExtractFeatures();
// Step 6: Feed features to
   classifier and get label
// Step 7: Handle label
if Label is RED then
    // Step 7a: Do not upload to FPGA
    Do not upload to FPGA;
end
else if Label is YELLOW then
    // Step 8a: Find suitable FPGA
    FindSuitableFPGA();
    // Step 8b: Upload to FPGA
    UploadToFPGA();
    // Step 8c: Alert online tool
    AlertOnlineTool();
end
else if Label is GREEN then
    // Step 9a: Upload to first fitting
       FPGA
    Upload to first fitting FPGA;
end
// Step 10: Return label
return Label;
```

TABLE II: Terminology used in Sections IV and V.

Term	Explanation
Basic Design	HDL code of one module, e.g., DES or JPEG
Tenant Design	One basic design or several of them in a cluster
Tenant region	Area on the FPGA assigned to one tenant
Floorplan	Partitioning the FPGA into different tenant regions
Bitstream	Tenant design in binary, uploaded on the FPGA

benchmarks, we use some of our developed basic designs, such as JPEG compression/decompression, SHA, and RSA. We mix and match the basic designs from these benchmarks to build the tenant designs. Table III shows all basic designs, the benchmarks they originate from, and the frequency of using them in our dataset. Accessing and using real tenant designs from CSPs is not possible. Even though AWS Marketplace [41] provides FPGA cores, they are typically either simple IP cores meant for integration into larger designs [42], or they are complete systems running in software that uses hardware IPs. The complete systems utilize hardware accelerators through an interface without direct access to the tenant design itself [43]. Therefore, we rely on benchmarks as done by [31], [30] to

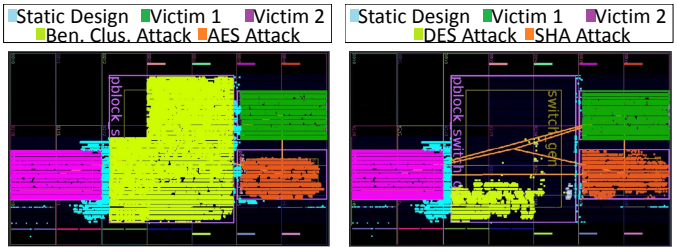
fulfill our evaluation, covering a range of applications suitable for FPGA acceleration, including Neural Networks and Bitcoin mining.

TABLE III: Basic Designs for Bitstream Generation

Basic Design	Benchmark	#Bitstreams
JPEG	Own Designs	61
RISCV	RISC-V River SoC	15
AVA decoder	Berkeley	42
RSA	Own Designs	46
Cluster of seq. circuits	ISCAS Sequential	64
Serial keyboard	Groundhog	24
PID Controller	Groundhog	45
FIR	Berkeley	28
FFT	Groundhog	40
Bitcoin miner	Opencores	22
AES Attack	Attack from [18]	59
Mux Attack	Attack from [15]	5
Shift register attack	Attack from [18]	20
RAM Attack	Attack from [27]	19
Reed-Solomon attack	Own Designs	19
DES*	Berkeley	25
SHA*	Own Designs	25
Glitch Attack	Attack from [28]	20
Latch Attack	Attack from [15]	20
Neural Network	Opencores	38
Ethernet	Opencores	38
CRC	Opencores	57
SPI	Opencores	57
Manchester encoder	Opencores	38
IIR	Opencores	38
DCT	HLS	25

* Used both maliciously and benignly

We generate bitstreams for the ZCU102 FPGA board, utilizing its Xilinx UltraScale+ FPGA for measurements to establish labeling ground truths. These bitstreams are then loaded onto the FPGA board. Our focus lies in detecting the success of attacks, which determines the labeling of the bitstreams. The same bitstreams can be used across multiple target FPGA boards, mirroring a cloud scenario from the user's perspective.



(a) AES and benign cluster coordinated tenant attacks

(b) SHA and DES coordinated tenant attacks

Fig. 3: Floor-planning of tenants where multiple tenants have different resource assignments and utilization on the same FPGA.

A. Generating the Tenant Designs

We employed various strategies to create tenant regions. For example, Fig. 3 demonstrates the implementation of coordinated attacks from multiple tenants (see Section II-B). The FPGA's floor plan is divided into four regions, with two hosting malicious designs and the other two hosting benign ones. One region utilizes 50% of the resources, while the other three each utilize 15%, leaving 9% for the static design. In the example shown in Fig. 3, the 50% region is positioned in the middle of the FPGA. However, for another floor plan, the 50% region can be placed at the top or bottom of the floor plan, not necessarily in the middle. This contributes to diversifying the bitstreams by avoiding constraining them into fixed regions but instead across several different regions.

A CSP typically maintains several floor plans to accommodate various types of users. For instance, the 50% tenant region from Fig. 3 can be substituted with two smaller tenant regions, each utilizing 25% of the resources. We employed six different floor plans to generate 24 distinct tenant regions for placing tenant designs. The sizes of these regions vary, ranging from 50% of the FPGA resources to 15% of the FPGA resources.

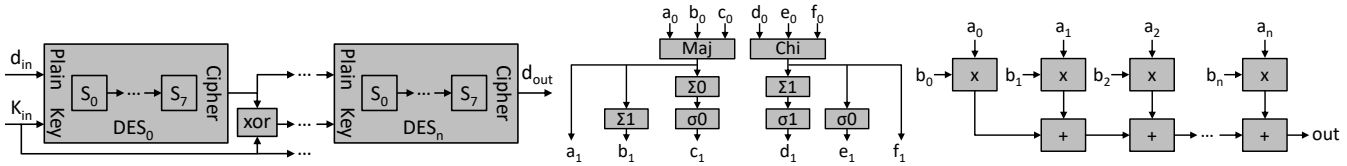
Not all tenant designs were used in all the tenant regions as they might not fit into them, i.e., they need more resources than the region provides. Those tenant designs that did not fit were either modified, like changing the RISC-V dual core to a single core, or we diversified the designs further by the following modifications: (i) mixing them more, e.g., substituting a large FFT module with a smaller PID-controller module and a Manchester encoder, (ii) increasing the repetition within the design, e.g., adding multiple JPEG compression instances after removing a large DES module. Moreover, we hide some malicious modules with benign modules making the attacks stealthier similar to [30]. The generated tenant designs are categorized into 153 GREEN ones, 120 RED ones, and 177 YELLOW ones, as detailed in Section III-D.

B. Implementation of Attacks based on Benign Constructs

We generated malicious tenant designs similar to the AES malicious design from [18] to enrich the dataset. These malicious tenant designs are based on the Data Encryption Standard (DES), Secure Hash Algorithm (SHA), and Reed-Solomon, as depicted in Fig. 4. The malicious DES-based design in Fig. 4a utilizes unrolled DES S-boxes as the fundamental building block. Multiple blocks are interconnected in a chain with adjustable chain lengths to fit the size of the tenant region. The output of each block serves as the input for the subsequent block. The key for each block is computed by XORing the output of the preceding block with the original key. This process amplifies the toggling along the path, thereby increasing the power consumption.

The malicious SHA-based design also employs a chain of interconnected SHA sub-functions (shown in Fig. 4b). Each sub-function receives six inputs, which are mixed to produce the various components of the SHA algorithm, resulting in six outputs. The output of one sub-function can be directly connected to the next's input, with the chain's length configurable as desired. Note that only the first input originates from registers, and no combinational loops are present in the design.

As the Reed-Solomon encoder inherently comprises a chain of multiply-accumulate operations, the registers between the adder stages are simply removed to transform it into a malicious design (see Fig. 4c). This modification results in a lengthy combinational path, which can be configured as desired. The inputs originate from tenant-internal registers initialized by constants and subsequently inverted in every cycle to enhance toggling.



(a) DES-based attack (b) SHA-based attack (c) Reed-Solomon-based attack

Fig. 4: Implemented attacks, derived from benign modules. With small modifications, removing sequential elements, and special toggling input patterns, they lead to successful attacks

Furthermore, to enhance the difficulty of detection, we explore the concept of hiding these malicious designs among benign ones to evade detection by current state-of-the-art solutions. We integrate the malicious designs alongside a cluster of ISCAS sequential circuits [36]. Consequently, a bitstream scanner would identify slightly modified benign designs and encounter additional circuits introducing randomness to the structural design. This combined setup presents a more complicated functionality resembling a standard design, performing tasks beyond solely cryptographic operations or encoding.

V. Evaluation

We implement the tenant designs using Vivado 2019.1 to evaluate our proposed Meta-Scanner. The bitstreams were uploaded to a ZCU102 board. Meta-Scanner is implemented in python and tested on an AMD Ryzen 5 6-Core processor with 24 GiB main memory.

A. Ground Truth of Benign-based Attacks

To label the malicious tenant designs from Section IV-B we run them on a ZCU102 board to see if they crash the FPGA. Table IV shows the results. Utilization (%) is based on the total LUTs available in the ZCU102 FPGA board. Any version of the malicious designs having the size from Table IV or larger are labeled as RED.

Furthermore, we classify smaller malicious designs as YELLOW due to their potential for coordinating attacks, substantiated by the findings presented in Table IV. Initially, when both tenants, SHA and DES, are malicious and deploy weakened versions of their attacks, a coordinated attack becomes feasible. Secondly, in scenarios where only one tenant (AES) is malicious but cannot execute an attack independently, it can exploit the presence of a resource-intensive benign tenant. When executed concurrently, the benign tenant inadvertently facilitates an attack, resulting in a system crash. Consequently, any benign large design capable of instigating an attack when combined with the small AES attack is classified as "YELLOW." It should be noted that in Table IV we show the speed of a crash for the minimum area. However, using more FPGA resources would cause faster attacks [10]. Moreover, LoopBreaker [10] can stop an attack fast only with pre-selection of the malicious tenant. Without our tool, LoopBreaker will not be able to identify the malicious-tenant and would need the lengthy selection step which requires hundreds of microseconds which is enough for almost all the attacks to succeed.

TABLE IV: Minimum time and utilization needed for achieving crashes using benign-based attacks.

Attack based on	Crash speed	Crash FPGA utilization
AES* [18]	12 μs	18.5%
Reed-Solomon*	167 μs	38.7%
DES*	90 μs	27.0%
SHA*	34 μs	21.9%
SHA + DES$^+$	60 μs	14.6% + 18.0%
AES + benign cluster$^+$	>2 Min	13.9% + 34.0%

* attack from single tenant
$^+$ attack from multiple coordinated tenants

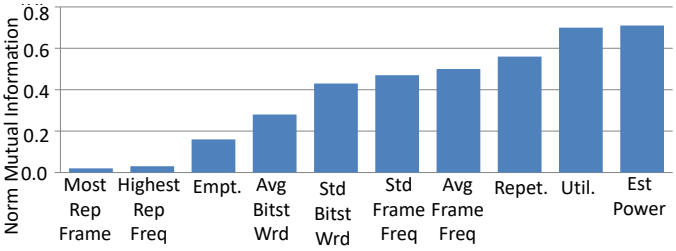


Fig. 5: Normalized Mutual Information of each feature individually to the data. 1 is maximum correlation and 0 means no correlation at all

B. Metadata Features' Importance

As mentioned in Section III-C, we use the scikit-learn library [33] to evaluate the relevance of our features to the classification problem. In Fig. 5, we show the Normalized Mutual Information (NMI) between each feature and the data before classification. NMI is one of the metrics from the scikit-learn library [33]. It assesses a normalized value with 1 being the highest value (the feature is very relevant to the classification problem) and 0 being the lowest value (the feature is not relevant at all to the classification problem). Utilization and Estimated Power have the highest NMI values of around 0.7. The other three features have NMI values around 0.5. The figure shows that all the metadata features relate significantly to the classes. Hence, they are all relevant to the classification problem and can correctly classify the tenant designs. For the five excluded features mentioned in Section III-C the NMI score is less than 0.5. Therefore, they are less suitable for the classification problem and excluding them is sensible.

Attacks rely on the malicious basic designs, e.g., AES-based, RAM-based, etc. Therefore, we further evaluate their importance per basic design. To be able to perform this evaluation, we had to use a one-class classifier method [44]. One-class classifiers belong to the unsupervised learning approach, where data from only one class is used for training. The result

TABLE V: Feature Importance per basic design when used in a one classifier model.

Basic Design	AvgFrameFreq	Repetition	EstimatedPower	StdFrameFreq	Utilization
Artificial Neural Network	2.7%	41.8%	8.9%	18.2%	28.4%
FFT	17.0%	24.0%	15.0%	27.0%	17.0%
RSA	39.4%	11.4%	8.4%	30.6%	10.2%
JPEG	6.9%	32.4%	27.0%	5.6%	28.1%
IIR	13.4%	20.3%	31.1%	15.3%	19.9%
Cluster of s13207/s1494/s9234	14.2%	27.6%	28.0%	7.5%	22.7%
Bitcoin Miner	18.9%	29.4%	16.9%	7.1%	27.7%
Serial keyboard	25.8%	15.8%	16.8%	14.8%	26.8%
SPI	1.0%	45.7%	13.4%	6.3%	33.6%
PID Controller	7.9%	11.8%	41.5%	12.8%	26.0%
DES*	15.4%	11.3%	42.7%	18.9%	11.7%
FIR	28.2%	9.7%	24.3%	18.6%	19.2%
Manchester Encoder	1.0%	36.2%	27.5%	9.8%	25.5%
SHA*	22.0%	6.4%	39.9%	24.6%	7.1%
AES Attack	8.1%	17.1%	44.0%	1.0%	29.8%
AVA decoder	13.5%	19.3%	31.3%	15.6%	20.3%
Ethernet	13.4%	20.3%	31.1%	15.3%	19.9%
RISCV	13.0%	19.8%	31.1%	14.5%	21.6%
Reed Solomon Attack	2.6%	37.3%	28.6%	4.7%	26.9%
CRC	14.1%	21.6%	31.2%	14.1%	19.0%
Latch Attacks	14.7%	14.5%	28.8%	19.9%	22.1%
RAM Attack	12.7%	39.1%	4.5%	29.1%	14.6%
Glitch Attack	19.0%	30.0%	0.0%	32.0%	19.5%
Shift Register Attacks	12.0%	18.6%	12.2%	39.4%	17.8%
Mux Attacks	0.5%	36.2%	9.8%	27.5%	26.0%
DCT	15.3%	26.2%	27.4%	19.2%	11.9%
Average	**13.6%**	**23.6%**	**25.1%**	**17.3%**	**20.1%**

* Used both as an attack and as normal module

of the classification is a binary true or false. For example, if we train a one-class classifier on the DES malicious designs, it will detect and label them as *true*. Anything else, even AES or Reed Solomon malicious designs, would be labeled as *false*.

The results are presented in Table V. Notably, no feature scored 50% or higher in importance across all cases. Exceptions were observed where the relevance of features varied among different basic designs. For instance, the Repetition feature held the highest importance for the Reed-Solomon malicious design with a score of 37.3%. Conversely, for the AES malicious design, the Power Estimation feature scored 44.0%, while Utilization scored 25.5%. Additionally, in cases such as RSA, Serial Keyboard, and FIR, the Average Frame Frequency feature, typically of low importance, exhibited significant relevance. Similarly, estimated power was less critical for ANN and Bitcoin miner designs, with repetition playing a more substantial role. Furthermore, estimated power played a minor role for several malicious designs like RAM, glitch, Mux, and shift registers, reaching 0.0% for glitch malicious designs. For these malicious designs, the standard deviation of frame frequency gained importance.

C. Performance of the Classifier

TABLE VI: Results of 10-Fold Cross Validation across 475 Total Bitstreams. Mean Accuracy: 0.979 ± 0.02. Mean accuracy of detecting newly introduced attacks: 0.968.

class	precision	recall	f1score	support	FPR	FNR
GREEN	0.990	0.979	0.984	17.8	0.007	0.020
YELLOW	0.969	0.978	0.972	17.7	0.015	0.016
RED	0.977	0.985	0.979	12.0	0.008	0.021
New RED	1.0	0.963	0.978	1.7	0.000	0.018

The metadata extracted from the bitstream generation is used to train the random forest classifier, as described in Section III-D. The training and test data are split randomly by having 10% of the data for testing and the rest as training data. The split is done using the split method from the scikit-learn library [33]. Additionally, we perform 10-fold cross-validation using our 475 bitstreams, and the results are shown in Table VI. The RED class has the highest recall and precision to avoid banning legitimate designs and not uploading malicious designs (achieved by fine-tuning the class weights as explained in Section III-D). The other two classes (GREEN and YELLOW) still have high precision and recall and the whole classifier has a mean accuracy of 0.979. Moreover, we ran inference on malicious designs based on our designs from Section IV-B. It had a mean accuracy of 0.95, a precision of 1.0, and a recall of 0.963. For false negatives and false positives, Table VI shows that the FPR and FNR are at highest of the value 0.021 which is comparably low. The FPR of the YELLOW class is roughly the double of the other two classes. This is due to the fact that it is the class in the middle, therefore, a RED design will most likely be misclassified as YELLOW and same for GREEN. For FNR, it is slightly lower for the YELLOW class than the other two, but in general, it stays low for all three classes.

Table VII compares our scanner against the five state-of-the-art approaches [14], [15], [16], [11], [30]. As they can only classify into two classes (attack vs. no attack), we decided to consider YELLOW and GREEN classes as 'no attack', to give them an advantage and to have a conservative comparison. Still, all state-of-the-art approaches have significantly lower accuracy compared to our scanner. Note that for the tools from [11], [30] the tool does not even support partial bitstreams in its current format. However, for a fair comparison, we assume they could be updated to support them. Our scanner is the only tool that detects BRAM short circuit malicious de-

TABLE VII: Comparing our solution to the state of the art. The numbers are based on our dataset. The types of attacks that each tool mentions that it can detect are considered correctly classified by the tool. The comparison is conservative to give the tools the highest possible fair score. We do not consider the accuracy mentioned in their papers but consider them perfectly capable of detecting any attack they mention they can detect. Only for our classifier, we present the mean accuracy $\pm$ the standard deviation but not for the other tools.

Metric	Ours	Ref. [14]	Ref. [15]	Ref. [16]	Ref. [11]	Ref. [30]	Ref. [31]
Accuracy	0.979 ± 0.02	0.789	0.756	0.709	0.840	0.825	0.836
Hidden Attacks	✓	✗	✗	✗	✓	✗	✓
Partial Bitstreams	✓	✓	✓	✓	✗	✗	✓
Cryptographic Bening-based Attacks	✓	✗	✗	✗	✗	✓	✓
Non-Cryptographic Benign-based Attacks	✓	✗	✗	✗	✗	✗	✗
Short circuit Attacks	✓	✗	✗	✗	✗	✗	✗
Coordinated Attacks	✓	✗	✗	✗	✗	✗	✗

signs and non-cryptographic benign-based malicious designs (Reed-Solomon-based and shift-register-based).

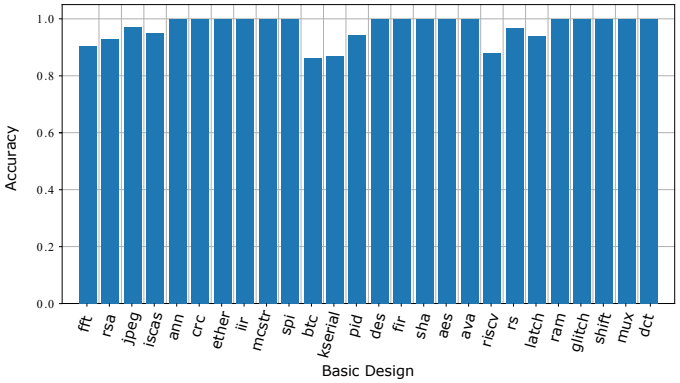


Fig. 6: Mean detection accuracy depending on basic designs.

Moreover, Fig. 6 shows the accuracy of classifying each basic design to the correct classes. The accuracy is defined as the number of samples correctly classified, divided by the total number of samples used for the inference. Many of the accuracy values are at 1.0, which means that no false positives nor false negatives occur for this basic design. Overall, all accuracy values are higher than 0.85. DES and SHA (which are both used as benign designs as well as malicious designs hidden using ISCAS circuits) have a high accuracy of 1.0. Hence, our scanner was able to correctly detect hidden malicious designs, and differentiate between using a module for an attack or using it as a true benign design. Moreover, our scanner can detect all the new malicious designs with high accuracy.

Additionally, we evaluate our timing overhead. As mentioned in Section III-A, the CSP performs Place & Route, feature extraction from the metadata, and scanning (inference of the classifier). Table VIII shows the results of running our scanner on the AMD Ryzen 5 6-Core processor with 24 GiB main memory. On average, Place & Route for one bitstream needed 27 minutes, while our feature extraction needs less than two seconds and the inference needs less than 10 milliseconds. Hence, our feature extraction and inference have negligible overhead. The feature extraction takes more time than the inference as it needs to parse the bitstream frame by frame. Moreover, we also measure the time needed for training, our solution needs on average 2 minutes to train the decision tree.

D. Performance against Unseen Designs

To complement the classical validation from Section V-C, we use an additional training and test strategy to evaluate

TABLE VIII: Timing overhead of our Classifier

	Training	Feature Extract	Inference	P&R
Time needed	2 min	1.6 s	5.0 ms	27 min

the generalization of our classifier. For each basic design b, we perform a training/evaluation experiment, declaring b as "unseen basic design" and excluding all bitstreams from the training phase that contains b. This mimics the scenario where a new malicious or benign design emerges that has been used for training. The not-excluded bitstreams are all used to train the model and we test the performance based on the excluded bitstreams. Note that this evaluation against unseen designs is not performed by any state-of-the-art solution [31], [30], [16], [11]. However, we decided to perform it as an extra step to evaluate the robustness of our scanner.

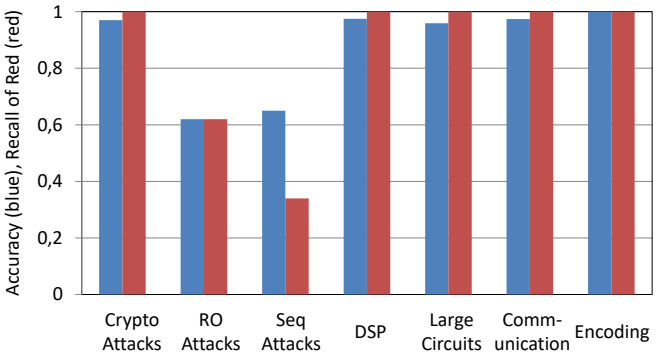
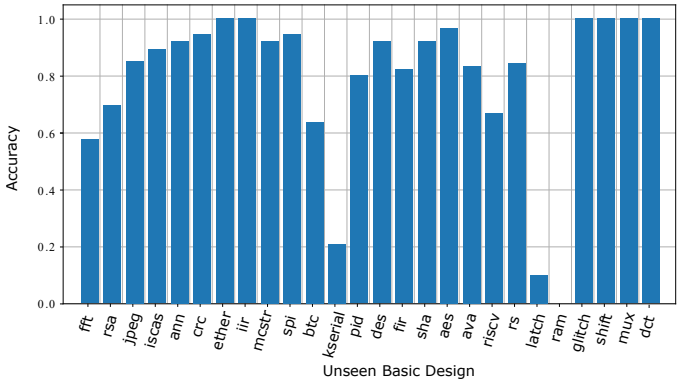


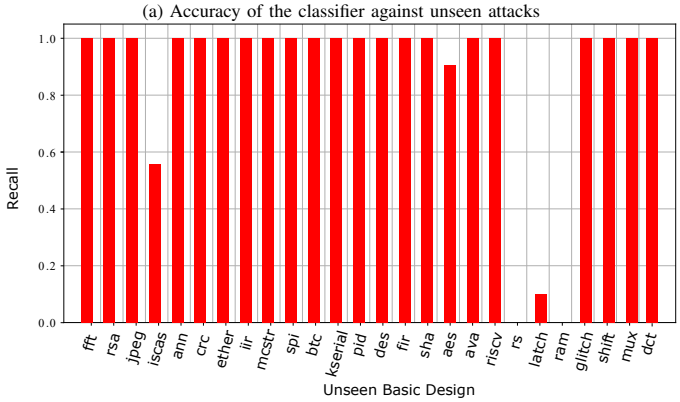
Fig. 7: Accuracy and recall per unseen category.

First, we start by evaluating the case where a full category of designs is unknown. For example, if no cryptography-based attacks were ever used before or if large circuits like neural networks or bitcoin miners are not used before. Figure 7 shows our results. For most categories, neither recall nor accuracy dropped under 0.9. However, for RO-attacks that use muxes and latches or sequential attacks that use RAM or reed-solomon encoder the accuracy and recall drop. The reason is that these attacks look very similar to benign small attacks.

We extend our analysis to be even more fine-grained. We do it per basic block level Figure 8 shows (a) the accuracy and (b) recall of the RED class for the different unseen basic designs. Recall of the RED class is of significant importance, as it shows how well our scanner stops malicious designs from being uploaded. It can be seen from the figures that the scanner's performance is adequate against the unseen designs



(a) Accuracy of the classifier against unseen attacks



(b) Recall of the RED class of the unseen attacks

Fig. 8: Performance of the classifier against unseen attacks. Bitstreams that contain an "unseen basic design" are not used for training.

with many of them nearly reaching the ideal value of 1 for both accuracy and recall.

However, there are some outliers. The outliers are analyzed and explained in the following, i.e., the Reed-Solomon malicious design, latch malicious design, RAM malicious design, keyboard serial, and the cluster of ISCAS benchmarks. The accuracy and recall are very low for the RAM- and latch-based malicious designs because they are the only malicious designs that do not use any LUTs. Both malicious designs can be implemented using only RAMs or latches, respectively. When completely excluding them from training, the classifier does not have any preceding knowledge that using only these resources can realize an attack. To counter that, all basic primitives that can be used for attacks must be included to train the classifier. This is conceptually the same as we did in the normal training of our classifier, as evaluated in Section V-C.

Similarly, the Reed-Solomon malicious design is the only one that neither uses high repetition of primitives nor cryptographic primitives. Instead, it is the only malicious design in our collection that uses long chains of combinational paths. Similar to the case of unseen RAM- and latch-based malicious designs, the classifier does not have any hint or knowledge that such a structure may cause a risk.

Only the RED recall is low for the cluster of ISCAS benchmarks, but the general accuracy is good. This is because it is used as the irregularity that hides the malicious designs. Thus, without using it, the classifier does not know that such hidden attacks exist and have a suboptimal recall.

Finally, for the serial keyboard, the accuracy is also pretty low. This is because it is used in several GREEN designs. Without having it, the classifier only sees clusters of different YELLOW class modules and will classify the GREEN as YELLOW. Note that the recall on RED is still 1.0, so no security degradation occurs.

These results show that using tenant designs with a high diversity for training is helpful. In a real-world deployment of the classifier, it should be continuously updated (retrained) with bitstreams from real usage on the CSP. The possibility of fast and easy retraining of our Decision Tree classifier (see Table VIII) highlights the important flexibility of our approach. As no offline bitstream checker can always perfectly separate malicious from benign designs (including malicious designs that have not been discovered yet), our proposed classifier can be easily and automatically adapted to any new malicious designs. More importantly, if a new malicious design is based on a similar concept to a previously known malicious design, e.g., using DES or SHA instead of AES, our classifier can detect it with high accuracy even when no retraining is performed, as seen in Fig. 8. When new types of malicious designs are detected or reported, the CSP simply needs to retrain the lightweight ML model and it can achieve high security again. The new malicious bitstreams are simply added to the training pool to perform the retraining.

VI. Discussion

One struggle that we faced during our work is the benchmark to evaluate our solution. Unfortunately, to the best of our knowledge, no cloud-based FPGA benchmark exists. A cloud-based FPGA benchmark could have helped to evaluate our classifier more accurately. Moreover, as we mention in Section IV, the available examples from commercial CSPs like AWS were not feasible to use as a benchmark. Hence, we had to build our benchmarking setup based on the same FPGA benchmarks used by the state-of-the-art [16], [11].

The problem that clients currently have to entrust their unencrypted design netlists to the CSP for verifying the absence of potentially malicious circuits is an important ongoing research topic. In [32], a trusted attestation scheme is proposed, which could also be applied in our scenario, allowing the client to upload encrypted bitstreams together with a trusted shell for the CSP to verify the bitstream classification.

During our evaluation in Section V-D, we noticed outliers in detection for malicious designs not represented in other bitstreams. These instances were undetected until included in the training. This mirrors real-world scenarios where a new malicious design category may emerge initially undetected but can be added to training for subsequent detection. Furthermore, excluding specific malicious designs while including others from the same category led to successful detection, exemplified by AES malicious designs being detected due to the inclusion of other cryptographic core malicious designs like DES or SHA.

For retraining, we stand in contrast with the state-of-the-art as they are either very hard to retrain, e.g., the tools from [15], [29] that need to add the new designs and their

rules manually, or the tools from [16], [11] that only support self-oscillating structures and therefore cannot be seen as applicable against more advanced attacks. The only promising tools are from [30], [31] but both mention nothing about their retraining. For the tool from [30] it is not open source so we could not evaluate it but for the tool from [31] it is open source so we evaluated it. They use a parser for simulation netlists which needs to be updated and maintained for each new type of circuits which is a significant overhead not only for training but as an engineering effort. Our tool works directly on the bitstream and does not require any special maintenance.

VII. Conclusion

In this work, we proposed an Meta-Scanner, a tool for detecting fault attacks in multi-tenant cloud FPGA instances. We first analyze the bitstream structure to extract relevant metadata based on them we implemented the classifier for our scanning scheme. By categorizing client bitstreams into three different risk classes through a machine learning approach, high-risk designs are prevented from being uploaded, whereas low-risk designs can be mapped to FPGA regions arbitrarily. Potential attack designs in the mid-risk class can be uploaded, but as long as only a single such design is mapped per FPGA chip, they can be dealt with by existing on-chip countermeasures. Evaluating a random forest classifier on a comprehensive set of 475 different malicious and non-malicious bitstreams leads to an overall average classification accuracy of 0.979 ± 0.02, proving the feasibility of our proposed approach. Our solution has a low overhead for training and scanning (inference). Moreover, it can be easily adapted to any new emerging type of attack.

References

[1] Amazon Web Services (AWS). (2021) EC2 F1 instances. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[2] Intel Corporation. (2021) Intel FPGAs power acceleration-as-a-service for alibaba cloud. [Online]. Available: https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud

[3] Huawei Cloud. (2021) FPGA accelerated cloud server. [Online]. Available: https://www.huaweicloud.com/en-us/product/fcs.html

[4] Open Telekom Cloud. (2021) Faster computing: FPGA hardware acceleration for the open telekom cloud. [Online]. Available: https://open-telekom-cloud.com/en/blog/product-news/fpga-closed-beta

[5] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing", in *CloudCom*, 2015.

[6] Y. Luo and X. Xu, "HILL: A hardware isolation framework against information leakage on multi-tenant FPGA long-wires", in *ICFPT*, 2019.

[7] G. Dessouky, A.-R. Sadeghi, and S. Zeitouni, "SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities", in *IEEE EuroS&P)*, 2021.

[8] M. G. Jordan, G. Korol, M. B. Rutzig, and A. C. S. Beck, "MUTECO: A Framework for Collaborative Allocation in CPU-FPGA Multi-tenant Environments", in *SBCCI*, 2021.

[9] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "FPGAhammer: remote voltage fault attacks on shared FPGAs, suitable for DFA on AES", *IACR TCHES*, 2018.

[10] H. Nassar, H. AlZughbi, D. Gnad, L. Bauer, M. Tahoori, and J. Henkel, "LoopBreaker: Disabling interconnects to mitigate voltage-based attacks in multi-tenant FPGAs", in *ICCAD*, 2021.

[11] R. Elnaggar, J. Chaudhuri, R. Karri, and K. Chakrabarty, "Learning malicious circuits in FPGA bitstreams", *IEEE TCAD*, 2022.

[12] D. R. E. Gnad, F. Oboril, and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams", in *FPL*, 2017.

[13] T. La, K. Pham, J. Powell, and D. Koch, "Denial-of-Service on FPGA-based Cloud Infrastructures — Attack and Defense", *IACR TCHES*, vol. 2021, 2021.

[14] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "Mitigating electrical-level attacks towards secure multi-tenant FPGAs in the cloud", *ACM TRETS*, 2019.

[15] T. M. La, K. Matas, N. Grunchevski, K. D. Pham, and D. Koch, "FPGADefender: Malicious self-oscillator scanning for Xilinx UltraScale+ FPGAs", *ACM TRETS*, 2020.

[16] J. Chaudhuri and K. Chakrabarty, "Detection of Malicious FPGA Bitstreams using CNN-Based Learning*", in *ETS*, 2022.

[17] G. Provelengios, D. Holcomb, and R. Tessier, "Mitigating voltage attacks in multi-tenant FPGAs", *ACM TRETS*, 2021.

[18] G. Provelengios, D. Holcomb, and R. Tessier, "Power wasting circuits for cloud FPGA attacks", in *FPL*, 2020.

[19] H. Nassar, P. Machauer, D. R. E. Gnad, L. Bauer, M. B. Tahoori, and J. Henkel, "Covert-hammer: Coordinating power-hammering on multi-tenant fpgas via covert channels", in *ACM/SIGDA ISFPGA*, 2024.

[20] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen, "FPGA Resource Pooling in Cloud Computing", *IEEE Transactions on Cloud Computing*, 2021.

[21] M. Paolino, S. Pinneterre, and D. Raho, "FPGA virtualization with accelerators overcommitment for network function virtualization", in *ReConFig*, 2017.

[22] *Introduction of F1 development environment*, Amazon Web Servises (AWS), 2021.

[23] H. Englund and N. Lindskog, "Secure acceleration on cloud-based FPGAs – FPGA enclaves", in *IEEE IPDPSW*, 2020.

[24] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "An inside job: Remote power analysis attacks on FPGAs", in *DATE*, 2018.

[25] T. Sugawara, K. Sakiyama, S. Nashimoto, D. Suzuki, and T. Nagatsuka, "Oscillator without a combinatorial loop and its threat to FPGA in data centre", *IET Electronics Letters*, 2019.

[26] A. Boutros, M. Hall, N. Papernot, and V. Betz, "Neighbors from hell: Voltage attacks against deep learning accelerators on multi-tenant FPGAs", in *ICFPT*, 2020.

[27] M. M. Alam, S. Tajik, F. Ganji, M. Tehranipoor, and D. Forte, "RAM-Jam: Remote temperature and voltage fault attack on FPGAs using memory collisions", in *FDTC*, 2019.

[28] K. Matas, T. M. La, K. D. Pham, and D. Koch, "Power-hammering through glitch amplification – attacks and mitigation", in *FCCM*, 2020.

[29] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "Remote and stealthy fault attacks on virtualized FPGAs", in *DATE*, 2021.

[30] J. Chaudhuri and K. Chakrabarty, "Diagnosis of malicious bitstreams in cloud computing FPGAs", *IEEE TCAD*, 2023.

[31] L. Alrahis, H. Nassar, J. Krautter, D. Gnad, L. Bauer, J. Henkel, and M. Tahoori, "Malignnoma: Gnn-based malicious circuit classifier for secure cloud fpgas", in *IEEE HOST*, 2024.

[32] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, "Trusted configuration in cloud FPGAs", in *FCCM*, 2021.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, 2011.

[34] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2009.

[35] P. Jamieson, T. Becker, P. Y. K. Cheung, W. Luk, T. Rissa, and T. Pitkänen, "Benchmarking and evaluating reconfigurable architectures targeting the mobile domain", *ACM TODAES*, 2010.

[36] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits", in *ISCAS*, 1989.

[37] OpenCores. (1999) Opencores the reference community for free and open source gateware ip cores. [Online]. Available: https://opencores.org

[38] J. Pistorius, M. Hutton, A. Mishchenko, and R. Brayton, "Benchmarking method and designs targeting logic synthesis for fpgas", in *IWLS*, 2007.

[39] *Vitis HLS Tutorial (v2023.1)*, Xilinx, Inc., 2023.

[40] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set manual", 2014.

[41] I. Amazon.com. (2022) Aws marketplace. [Online]. Available: https://aws.amazon.com/marketplace

[42] S. for Economists. (2022) Vfi - accelerator fpga. [Online]. Available: https://aws.amazon.com/marketplace/pp/prodview-fboutdmufjkum

[43] Huxelerate. (2022) Hugenomic nanopolish.

[44] S. S. Khan and M. G. Madden, "One-class classification: taxonomy of study and review of techniques", *The Knowledge Engineering Review*, 2014.