PhasePrint: Exposing Cloud FPGA Fingerprints by Inducing Timing Faults at Runtime

Jubayer Mahmod*
Virginia Tech
Blacksburg, USA
jubayer@vt.edu

Matthew Hicks
Virginia Tech
Blacksburg, USA
mdhicks2@vt.edu

Abstract

Cloud FPGAs, with their scalable and flexible nature, are rapidly gaining traction as go-to hardware acceleration platforms for compute-intensive workloads. However, their increasing adoption introduces unique security challenges. The hardware-level access that FPGAs provide leads to many vulnerabilities, including the leakage of sensitive information through data remanence and the creation of analog-domain covert channels among users. A foundational requirement in these scenarios is the ability to target an individual FPGA; knowing this, cloud vendors prevent FPGA localization by restricting access to low-level information of the underlying hardware. Beyond aiding adversaries, FPGA localization enables defenders to strategically rotate FPGA usage, preventing prolonged exposure that can lead to confidential data leakage due to long-term data remanence.

This paper introduces *PhasePrint*, a cloud FPGA localization approach using dynamic timing faults in functionally valid circuits. *PhasePrint* induces timing faults in a specially crafted circuit at runtime and infers delay characteristics from the resulting error pattern—without incorporating information sources blocked by cloud vendors. *PhasePrint* utilizes an FPGA's internal clock synthesizer to derive a clock pair with a strict phase relationship. By adjusting the phase relationship of these clocks, *PhasePrint* intentionally causes timing faults at runtime that reveal manufacturing variations among FPGA chips. We transform these fault locations into feature vectors to create device signatures and train a multi-class classifier on a dataset from 300 unique FPGAs across four AWS geographic regions. This entirely **on-chip** signature extraction method achieves **>99% accuracy**, **operates 13× faster**, and **costs 92% less** than the state-of-the-art.

*This work was completed while the author was at Virginia Tech and is independent of his current employment at AWS Security, Seattle, USA.

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716012

CCS Concepts: • **Computer systems organization** → *Reconfigurable computing*; • **Security and privacy** → **Side-channel analysis and countermeasures**.

Keywords: Cloud FPGA security; FPGA localization; On-Chip device fingerprinting

ACM Reference Format:
Jubayer Mahmod and Matthew Hicks. 2025. *PhasePrint*: Exposing Cloud FPGA Fingerprints by Inducing Timing Faults at Runtime. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3676641.3716012

1 Introduction

Cloud FPGAs provide flexible, scalable, cost-effective solutions for many compute-intensive workloads, such as neural networks and video transcoding. Most major cloud providers, including AWS, Microsoft Azure, and Alibaba Cloud, offer on-demand FPGA rentals. These virtual machines typically run specialized operating systems with pre-installed FPGA development tools, enabling customers to design and deploy accelerators without expensive hardware investments or software licenses. While this shared hardware model democratizes access to hardware acceleration, it also introduces unique security challenges inherent to FPGA systems.

FPGAs grant unparalleled control over their applications, a trait that introduces security vulnerabilities absent in conventional compute instances. Risks of information leakage arise both from simultaneous use by multiple tenants [21] and from sequential use by single tenants [9]. For example, malicious circuits can disrupt co-tenant operations by inducing timing failures through instantaneous voltage drops [31]. Users can even establish covert communication across virtual machines by exploiting the shared power supply between FPGA boards running concurrently on the same server [18]. Despite current practice of assigning an FPGA to only one user, covert/side-channel attacks persist.

Effective exploitation of data remanence-based side channels [9], covert channels [37], or strategies to manipulate victims into using specific FPGA devices in the cloud *hinges* on an attacker's ability to identify specific FPGAs. Additionally, locating an FPGA in the cloud (i.e., identifying neighboring FPGAs) helps defenders assess the risk of data leakage

through power-side channels. This knowledge also plays a crucial role in evaluating the risk of data leakage due to analog domain remanence [9]. Being able to identify a specific FPGA in a set of devices provides defenders with a powerful capability: it enables them to devise strategies that reduce data burn-in.

Identifying these FPGA-specific challenges, cloud service providers restrict access to device-level information. They enforce numerous security-focused policies on hardware accelerators, *from* development-phase checks *to* runtime monitoring. For example, AWS monitors FPGA power consumption to ensure a malicious circuit does not draw more power beyond some threshold. When a circuit goes above the threshold, AWS gates the circuit's clock, preventing it from wasting power. In addition, cloud FPGA vendors restrict access to FPGA-identifying information like *eFuses* and device fingerprints [35] to prevent FPGA localization.

During the design phase, developers must submit their designs to AWS for verification and binary generation. This process, unlike on-premise development, removes developers from the binary generation stage. AWS enforces design rule checks to ensure that submitted designs follow guidelines and exclude malicious components, such as self-oscillating circuits, e.g., ring oscillators, as they can be weaponized by an adversary to violate vendor isolation and localization barriers. Additionally, user designs must use an AWS-provided hardware shell to interface with the host CPU and for its main clock source. The resulting FPGA image is then protected with read-only access permission. These measures prevent users from uncovering low-level information about an FPGA, effectively blocking any *spatiotemporal* device tracking [24].

Given the importance of FPGA localization, researchers have developed indirect FPGA fingerprinting approaches that comply with both runtime and design regulations. For example, identifying an AWS F1 instance is possible using on-board DRAM-based Physical Unclonable Functions (PUF) [38]. This technique uses AWS-provided DRAM data retention features across multiple FPGA images [6]. The step involves writing data into DRAM, allowing a discharge period, and finally reading back the values after a predefined no-refresh interval. The fingerprinting scheme depends on the current behavior of the DRAM data retention setup, where it keeps DRAM modules "on" even when an FPGA image contains *no* memory controller. Apart from off-chip board-level component dependency, the process is inherently slow as it requires sequential loading of three FPGA binaries to acquire an FPGA card's signature. These limitations highlight the need for a cloud FPGA localization method that efficiently identifies *chips* and overcomes these challenges.

Our approach transforms any circuit into a time-to-digital converter using dynamic timing modulation, removing the limitations of state-of-the-art. Our system leverages timing

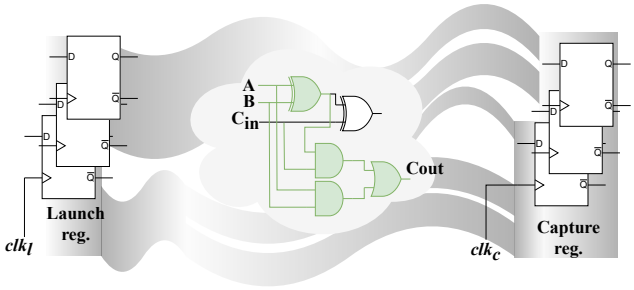


Figure 1. An illustration of some combinational logic between two registers. The signal propagate from a launch register at the positive edge of the launch clock (clk_l), gets modified by the logic, then is latched into the capture register at the positive edge of the capture clock (clk_c). Abstractly, you can imagine decreasing the phase delay of the capture clock compared to the launch clock from a full period (i.e., the clocks are the same) to 0 and noting the phase when the output becomes incorrect (i.e., a timing fault). *PhasePrint* leverages the insight that this phase is chip-specific, due mostly to manufacturing-time variation.

faults that cause computational errors at runtime by controlling a signal's propagation time through physical paths in the FPGA fabric. This process reveals device-specific manufacturing variation, which manifest as differences in propagation delay, eventually bubbling up from the analog to the digital domain as timing-error-influenced digital values. Upon this insight we design *PhasePrint*, a could-FPGA localization system that exposes the gaps in the design rule checks and runtime restrictions employed by cloud FPGA vendors and improves upon the state-of-the-art. *PhasePrint* serves as a foundation for both attacks and defenses that require FPGA localization in the cloud.

Our system's hardware component utilizes an FPGA's internal clock synthesizer to generate two synchronized clocks: launch and capture (Figure 1). The system feeds input into the circuit at the launch clock's positive edge and latches the outputs at the capture clock's positive edge. The circuit complies with design rules during the development phase and allows sufficient time for signal propagation from launch to capture registers. However, we utilize a runtime phase-shifting mechanism to narrow the interval between these clock edges, preventing the circuit input from fully propagating and stabilizing before the capture clock's arrival—causing computation errors. By maintaining clock phase alignment and adhering to the circuit's timing constraints, our approach circumvents the cloud vendor's security measures, including timing analysis [19].

We design a circuit that samples logical paths at various points to measure signal propagation distance within the launch and capture interval. Subsequently, we transform the timing failure points across these paths into distinctive feature vectors and then use them to develop a support vector machine-based multi-class classifier for FPGA identification.

Our classifier determines if an FPGA matches any within our baseline collection of 300 unique FPGAs sourced from four different AWS geographic regions in the United States and Europe.

In summary, this paper makes the following technical contributions:

- We introduce a method that leverages an FPGA's clock synthesizer to extract low-level information *without* the need for self-oscillating circuits (§5).
- We develop *PhasePrint*, a fully on-chip cloud FPGA signature extraction scheme that delivers over **a 13-fold faster performance** compared to the state-of-the-art and complies with stringent cloud-vendor security standards (§5.1, §5.2, and §6.4).
- We validate *PhasePrint* on a dataset comprising 300 unique FPGAs from four AWS geographic regions, achieving **over 99% accuracy** in FPGA detection and reducing the cost of signature extraction by **92%**(§6).
- 24-hour tests show that *PhasePrint* is robust against operational noise such as voltage and temperature fluctuations in the cloud (§6.3).
- We demonstrate that initiating a new set of FPGAs shortly after terminating a previous set results in an overlap exceeding 50% between the two sets of FPGA allocations (§6.5).
- We open source our *PhasePrint* implementation and supporting AWS instrumentation code [26].

Responsible Disclosure: In accordance with standard vulnerability disclosure procedures, we submitted a preliminary draft of this paper and associated artifacts to AWS Security in February 2024.

2 Background

Operating conditions, such as voltage and temperature, along with the manufacturing process, influence the speed at which signals propagate through a path. Maintaining constant operating conditions allows us to isolate delay variations caused by device-specific differences. This foundational principle also applies circuits running on an FPGA: running the same combinational circuit on two different FPGAs results in two different completion times for those FPGAs. Put a different way, if you had the ability to pause computation at any moment before completion, you would see the calculation is at different stages between devices— despite everything else being identical. Instead of pausing computation, you could overclock the circuit to expose partial progress of the computation. Thus, reducing the time allotted for computation below what it requires reveals device-specific characteristics. *PhasePrint* accomplishes this by configuring a Mixed Mode Clock Manager (MMCM) to expose this variation through a specially designed circuit, providing low-level, device-specific information to localize cloud FPGAs.

2.1 Cloud FPGA Design Flow

The FPGA development flow in the cloud deviates from traditional on-premise FPGA development, particularly in the final stage. Unlike traditional FPGA development, where developers generate a bitstream, cloud-based FPGA development requires them to upload a final design checkpoint instead. This checkpoint contains the fully-routed design and other critical information like clock configurations, adhering to a cloud vendor's design rules, and security measures. Using a vendor-provided hardware shell is a key feature in cloud FPGA designs. A shell establishes a connection between the hardware accelerator (*i.e.,* custom logic) and the host CPU via PCIe bus [5]. This shell offers ports, resets, and clocks to control and communicate with the user's custom logic. Cloud vendors implement stringent security measures at both the development and runtime stages to safeguard the integrity and confidentiality of user data.

2.1.1 Runtime monitoring: The runtime monitoring system ensures that a custom logic does not exceed a specified power budget and remains within a safe operating temperature limit [1]. If a design exceeds the power budget during loading, the shell triggers a violation flag and disables the clocks to custom logic, causing a shell timeout for all transactions [1]. To enforce these rules, AWS provides preconfigured clocks from the shell, which are fully controllable from the runtime monitoring system. In fact, AWS does not allow an arbitrary clock frequency in a circuit and only permits a set of clock recipes. This is an essential precaution because self-oscillating circuits are malicious to the AWS cloud infrastructure, and they can pave the attack surface leading to breach user confidentiality [14]. This is because these circuits can fingerprint the FPGA or draw an excessive amount of power to orchestrate denial of service attacks [16, 23].

2.1.2 Design-time checks: As mentioned, FPGA bitstream generation is beyond a developer's control, which ensures that the final design follows preset guidelines (known as design rule checks). Cloud vendors scan a design's *final checkpoint* to ensure it is free of malicious circuit structures such as ring oscillators. Furthermore, when an FPGA image generation is complete, it remains immutable in the cloud, and one needs to load the image onto an FPGA instance only using management tools, thwarting any alterations of the bitstream and reverse engineering attempts. These security precautions effectively prevent the extraction of low-level information, such as device DNA, from FPGAs.

Next, we briefly explain how an erroneous computation due to timing failure exposes a device's low-level information and allows us to infer the physical properties (*e.g.,* speed of a particular path) of an FPGA.

2.2 Device Dependency of Erroneous Computations

If a circuit operates with a clock period *shorter* than the time needed for all logic states to settle at their final values, it experiences timing failure, which manifests as computational errors. Designers set the maximum clock frequency for a circuit and include a minimum timing margin, commonly known as *slack*, to tolerate device manufacturing and runtime variations. For instance, in Figure 1, if the launching clock (clk_l) and capturing clock (clk_c) are the same, we expect the signals to propagate from left to right within *one* clock pulse. This implies that the sum of the propagation time (tp), the hold time of the launch registers, and the setup time of the capture registers is less than the clock period [39]. [1]. If this condition is not met at design time, timing analysis tools report timing failures on the violating paths. Designers mitigate these issues by employing *pipelining* in the combinational paths and/or by extending the time period (T).

A circuit that experiences timing failure computes the output based on the current logic states at different nodes during a clock edge. We explain this phenomenon further with an adder circuit as it is central to understanding *PhasePrint*. Let us assume that the previous computation leads to C_{out} to be 1 ($A = 1, B = 1, C_{in} = 1$) in the circuit in Figure 1. When new values ($A = 1, B = 0, C_{in} = 0$) arrive in the inputs at clk_l, C_{out} latches as 1 in the capture register if any of the *and gates* fail in $1 \rightarrow 0$ transition by clk_c's arrival, which is erroneous.

FPGA tools (*e.g.,* Vivado from Xilinx) identify paths that fail to meet timing requirements at different stages of development. These tools are capable of detecting timing issues at design time; therefore, an attempt to run circuits at excessively high speed to induce failed timing behavior is detectable [24, 28].

When an allowed propagation time (from launch to capture registers) is close to the critical path delay, leaving no timing margin, computation results become device-dependent (under steady operating conditions) due to variations in the gates' response times across different devices. Consequently, some devices fail to yield correct results while others succeed. **This discrepancy in the timing behavior (observed through computation) effectively unveils the unique fingerprint of each device.**

2.3 FPGA Clock Modules

FPGAs feature MMCMs, offering functionalities like new clock synthesis, deskew, and jitter filtering of main clock. Developers configure these modules using static data, which gets synthesized during the FPGA design process and remains fixed post-deployment. Dynamic reconfiguration registers store various clock states, allowing the design to switch between them, much like a state machine [41]. Timing analysis tools examine this data to check the timing constraints of a design. In addition to static clock generation, cloud FPGAs facilitate runtime clock configuration using Phase Shift (PS) interface. Unlike static configurations, the phase relations among different clocks must be zero at design time. One can modify these relations at runtime using the PS interface. *PhasePrint* leverages this unique capability to shorten the signal propagation window at runtime, thereby inducing timing failures (§4).

[1]Clock skew occurs when downstream registers receive the clock edge later than the transmitting clocks [39]. For simplicity, this explanation assumes an ideal case with no clock skew.

3 Threat Model

Building on prior research [32, 35, 38], we posit that the ability to uniquely identify an FPGA device is crucial for both attackers and defenders. Attackers aiming to exploit side channels, such as those resulting from device wear [9], or to establish covert channels, such as heat-based methods [37], need to identify FPGAs in the cloud. An identification scheme allows localization, which in turn facilitates high-accuracy and high-confidence side channel attacks and enables efficient cross-user covert communications. Conversely, defenders leverage device fingerprints as a risk mitigation method. Strategically rotating designs across various FPGAs reduces the accuracy and likelihood of side-channel attacks and cross-user information leakage. We assume that the speed (hence cost) of device fingerprint extraction is crucial in achieving optimal attack and defense strategies. That is, knowing the identity of the physical device allows defenders to periodically shuffle their custom logic among different FPGAs to prevent (or reduce) the imprinting of information on the physical chip.

We assume attackers are able to deploy their designs that comprise typical elements such as LUTs and clock synthesizers. However, we consider that cloud service providers restrict access to low-level hardware details and deny access to device identifiers. We also assume that cloud providers rigorously vet designs to eliminate combinational loops and block access to I/O, internal voltage/temperature sensors.

4 Tuning Phase to Induce Errors

As shown in Figure 2, when clk_l goes high the launch register is updated with a new value. As soon as this value changes, the combinational circuit begins to process the value to create a corresponding output value. When clk_c goes high, the capture register latches the combinational circuit's output value. The interval between clk_l and clk_c sets the propagation window for signals to traverse the circuit paths. In simple synchronous sequential circuits clk_l and clk_c are the same and static timing analysis informs the hardware designer what the minimum clock period is. When the circuit is overclocked or the phase difference between clk_l and clk_c creates too short a window for combinational logic to complete, timing errors create incorrect results.

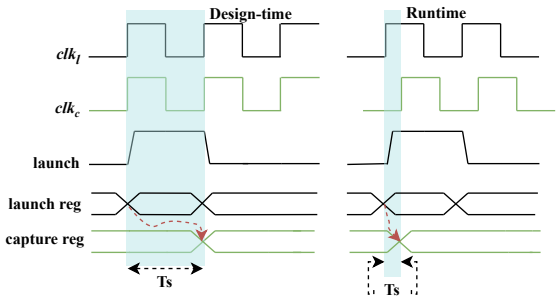


Figure 2. Propagation time modulation at design-time vs. runtime.

Figure 2 illustrates our modulation of this propagation window, deliberately inducing timing errors in a functionally valid circuit to expose analog domain variation in the digital domain. At design time, launch and capture *clocks are in phase* (Figure 2 left). At the positive edge of the launch clock (clk_l), we trigger the *launch* signal to inject the launch-register values *into* the combinational circuit. The input states propagate through different paths and transform into a final value according to the boolean function of the combinational circuit. At the positive edge of the capture clock (after the launch signal is initiated), the computed value gets captured in the capture register. As long as the time between launch and capture (T_s) is longer than the worst-case delay in any path (*i.e.,* critical path delay, T_c), the circuit experiences no timing failure, and the capture register latches correct values (§2). Figure 2 (left) shows a circuit operating with correct timing, where $Ts > T_c$ ensures the capture register latches the correct value, meeting the timing requirement.

However, errors emerge when we systematically (§2) reduce the propagation window at runtime by adjusting the phase relationship between the launch and capture clocks. Introducing a phase offset between these clocks satisfies the capture condition earlier (Figure 2 right), effectively shortening the available time for signal propagation. The following Verilog snippet demonstrates this condition.

```
always@(posedge clk_c) begin
    if(launch)
        capture_reg <= circuit_output;
end
```

The key takeaway is that dynamic phase tuning allows us to induce timing failures across various signal propagation paths within a design-time valid circuit. This results in incorrect outputs due to insufficient time for the logic states to settle. For a given propagation window, an output differs across FPGAs due to underlying manufacturing variations in the physical signal paths.

5 *PhasePrint* Design and Implementation

PhasePrint induces erroneous computation in a circuit at *runtime* and infers path delay characteristics to reveal process variation across different FPGAs. *PhasePrint*'s hardware component comprises a control unit configuring an MMCM and conditions internal signals to make a combinational circuit (*i.e.,* path delay extraction circuit) susceptible to timing faults. Meanwhile, the software component manages signal propagation timing and establishes communication between the host CPU and the attached FPGA(s). Figure 3 illustrates a high-level block diagram of our system. This section details *PhasePrint*'s hardware and software components and explains our data-driven approach for creating a device fingerprinting mechanism based on the concept in §4.

5.1 Hardware

The *PhasePrint* hardware consists of a vendor-provided shell and a Custom Logic (CL) block. This CL block includes our path delay extraction logic, a clock synthesizer module, a clock domain synchronizer, and additional control logic. The software component, running in the host CPU, interacts with the FPGA through an x16 PCI Express 3.0 and OCL interface. This software (§5.2) reads/writes data from/to PCIe-mapped registers, which the shell transfers to the CL.

5.1.1 Path delay extraction circuit: To expose process variation, we need to consider the path length and the clock synthesizer's minimum resolution (minimum phase difference between clk_l and clk_c). The dynamic phase shift mechanism allows the use of any circuit with paths that incur a long enough measurable delay. However, when designing a path for fingerprinting, it is essential to balance the minimum measurable delay with the maximum path length that exposes process information. A longer path may lead to timing failure but can mask process variation due to the averaging effect of the Law of Large Numbers. This happens when a path is long due to many physical components, and the overall delay becomes uniform across FPGAs. The random process variation averages out when many components are in a path, masking device-dependent characteristics [30]. For example, the minimum measurable time in Xilinx UltraScale+ FPGA is 11.16ps, when an MMCM is running at 1.6GHz–maximum frequency for the VCO [10]. The minimum step size is $\Delta ps = 1/(1.6G \times 56) \times 10^{12}$ [43]. Therefore, if a path requires longer than $11.16ps$ to propagate a logic value it is possible that it will create a measurable timing fault if we set $T_s = 11.16ps$.

We design device-specific process-variation-revealing combinational circuit paths using the CARRY8 elements of Xilinx Ultrascale+ devices. These elements, known for their consistent delay characteristics, are integral to the FPGA's architecture, facilitating fast carry propagation in arithmetic

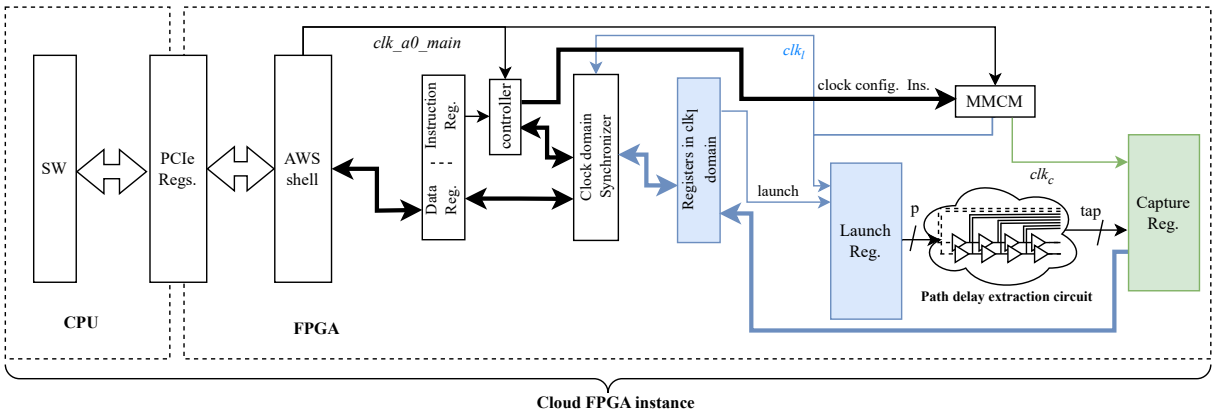


Figure 3. High-level block diagram of *PhasePrint*, illustrating its key components and signal flows. The diagram color-codes the clock domains: blue for clk_l, and green for clk_c. A clock domain synchronizer ensures the shell clock (`clock_a0_main`) is synchronized to these derived clock domains.

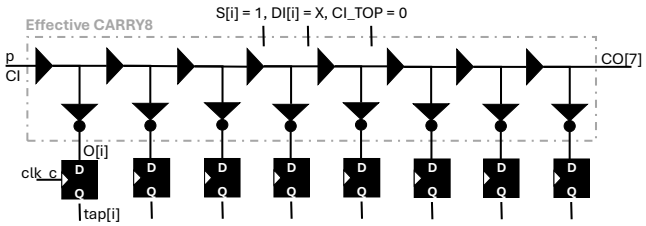


Figure 4. *PhasePrint* path delay sensor using CARRY8 elements. Paths longer than 8 bits are formed by connecting the carry in of a subsequent element to the carry out of the previous element.

operations such as addition and subtraction [34].[2] In addition, previous research explored these elements for various security applications, demonstrating their versatility and applicability in this context [8, 10, 47]. The limited logic between every tap into the carry chain (*i.e.,* a *MUX* and possibly an *XOR* gate) provides fine grain propagation delay information, which is strongly influenced by manufacturing-time process variation that is device specific [42]. We instantiate these elements directly in the Verilog hardware description language to design fingerprint extraction paths.

As shown in Figure 4, each CARRY8 element produces an 8-bit output, and, when configured as a carry chain, follows the equation: $tap[i] = S[i] \oplus C_{in}[i]$, where $i \in$ (0 to 7) and $S[i] = 1$, because that is the required value to cause the *MUX*es internal to the CARRY8 to form a carry chain. Thus, each CARRY8 forms a chain of eight parallel *NOT* gates (because an *XOR* with a constant-1 input has the same truth table as a *NOT* gate) connected by *MUX*es that act as buffers. The result of each gate is then exposed outside the CARRY8 element and stored in a flip-flop that acts as the capture register. To create paths longer than one CARRY8, we chain them together by connecting the carry output of one CARRY8 to the carry input of the next CARRY8. The launch register drives the carry input of the first CARRY8 in the chain. Note that the architecture guide provides more details about the construction of these elements and how to instantiate the CARRY8 primitive [42].

We initialize the delay line by driving it with the inverse logic value of the measurement value in order to stabilize the entire delay line at the complement of the desired value. This ensures accurate capturing of the extent of signal propagation. We also initialize the capture register to a value where every bit matches the measurement value as it will store the opposite value upon measurement due to the inverting behavior of the *XOR* gate. For example, if $p[0] = 0$ during launch, we expect *tap[i]* transition to 1 as the signal moves from left to right, thus we initialize the delay line by setting $p[0] = 1$ and $capture[0] = 8'h0$. This setup allows the same logical path to yield dual sources of information, as the state traverses two distinct physical paths depending on the value of *p[0]* and *S* (because a CMOS gate has both PMOS and NMOS transistors). Our *PhasePrint* implementation only uses one physical path per logical path as that alone is sufficient for accurate fingerprinting (§6). We arbitrarily set a path's initial value to 0 and measurement value to 1.

[2]Other FPGAs have similar carry chain logic that *PhasePrint* can leverage for deployments on those devices.

5.1.2 Controller: This unit decodes instructions written by the software in the instruction registers, activates various control signals, and ensures proper timing delay between clock configuration commands. The controller expects three instructions from the software (§5.2): 1) read/write data, 2) configure MMCM, and 3) launch inputs to the path delay extraction circuits (§5.1.1).

Figure 5. Bitmap visualization of timing failures in FPGA paths: how logic 1 (white) propagates through 32 paths in two different FPGAs (same design), changing the preset values (black) from left to right. Red dots indicate the first failing locations in each path.

5.1.3 Clock generation and fine phase shifting:

Our Custom Logic (CL) uses a 125MHz clock, `clk_main_a0`, provided by the AWS shell; this clock drives an MMCM to derive clocks (clk_l and clk_c same frequency for both) for the rest of our design. Most CL components run on clk_l except for the capture register, which is synchronized to clk_c.

We leverage the dynamic fine-phase shifting (PS) capability of the MMCM to modulate signal propagation time in the path-delay extraction circuit. Unlike state-machine-based configurations [41], this approach does not pre-set phase differences between clocks at design time, so cloud vendors are unable to ascertain the possible runtime phase relationships between the launch and capture clocks. This means that they cannot use simple timing analysis to defeat *PhasePrint*. At runtime, the controller sends clock configuration instructions to the MMCM's PS interface.[3]

5.1.4 Clock domain synchronization:

Since the system operates with multiple clock domains, we design a full duplex handshake mechanism to synchronize transactions across these domains. All read/write transactions begin with a request signal that originates in one clock domain and ends in another. A transaction is complete when the requesting domain receives an acknowledgment signal from the recipient clock domain. In this way, *PhasePrint* fully decouples the *shell clock* domain from the custom logic domain, allowing our custom logic to run with any clock frequency that we generate using the MMCM.

5.2 Software

PhasePrint's software component uses AWS FPGA management tools and libraries [3] to communicate with the FPGA from the virtual machine. In particular, we use PCIe-mapped register BAR0 as shared memory between hardware and software for sending control instructions and fetching data between CPU and FPGA [4].

To extract an FPGA's fingerprint, *PhasePrint*'s software executes the following steps:

1. Loads our Custom Logic (CL) into an attached FPGA.
2. Tunes MMCM to offset clk_c by a phase angle relative to the clk_l.
3. Transfers data to the CL's launch register by writing to the PCI-mapped BAR0 registers.
4. Triggers the input propagation through the path delay extraction circuit (§5.1.1).
5. Retrieves the analog-domain influenced result from CL's capture register for off-chip analysis.

We augment AWS's *FPGA development operating system* by adding the above software component to run fingerprint extraction service (*i.e.,* `systemd` service) in the f1 instance initialization phase [2]. This system service executes various operations, including initializing FPGA management tools and detecting the number of available FPGAs in a given instance. Upon detection of FPGAs in an instance, it starts fingerprint extraction processes automatically and concurrently for all available FPGAs on the PCIe bus. Upon completion, it records the fingerprint of the FPGA(s) and exposes it to user space for subsequent use, such as launching an attack or making strategic decisions regarding the deployment of a specific image on the FPGA.

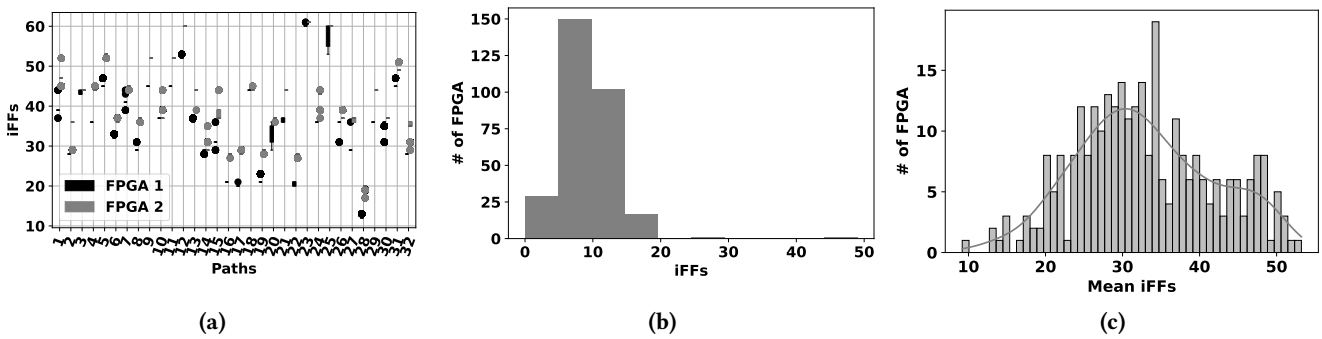
5.3 Concept of Operation

In normal circuit operation, if we initiate a signal propagation (*i.e.,* by setting p[0] = 0, as shown in Figure 4), the entire capture register flips from 0 to 1 at the positive edge of the capture clock (clk_c). However, the signal fails to traverse the entire path when we deliberately make this path susceptible to timing faults by tuning the propagation window ($T_s < T_c$). In this scenario, the capture register fails to complement its value in of the taps (Figure 5), resulting in incorrect computation like `11111100000.....`

5.4 Interpreting Failure Behavior

To interpret erroneous computation, we consider two codes: Hamming weight of the capture registers and the Index of the First Failure (iFF) of each path. Hamming weight provides the number of 1s in the capture register, allowing us to compute how far the signal has propagated within a particular T_s. This interpretation of the timing failure also includes transient noise, such as metastability, which is undesirable for fingerprinting applications [10]. To address this, we identify the first tap where a path outputs an incorrect value and then transform this failure index into a unary code [40]. Focusing solely on the first fault location minimizes the impact

[3] `clk_main_a0` clock drives PS interface clock (PSCLK) of the MMCM. The controller ensures a required 12 PSCLK delay between phase shift requests.



(a) (b) (c)

Figure 6. Demonstrating process variation as exposed by path delays: **(a)** failure location of the same circuit is FPGA dependent, **(b)** distribution of the failure locations of a single path across FPGAs, and **(c)** a distribution of the mean of 32 path's failure locations across FPGAs.

of transient noise. Thus, we quantify the signal propagation distance as an integer corresponding to the *tap*'s location (*i.e.,* index of First Failure), where we observe the first computational error (essentially failure to xor correctly). The iFF of the above pattern is 6 because the circuit outputs a wrong value in the 6th *tap* location (using zero-based indexing).

5.5 Experimental Demonstration

This section integrates the previously discussed concepts with experimental demonstrations, showcasing how the path delay extraction circuit (§5.1.1) combined with our runtime signal propagation control mechanism (§5.1) uncovers the manufacturing variations of an FPGA. First, we replicate a CARRY8-based path to capture intra-device variation and implement the essential logic (Figure 3) to execute dynamic timing faults in the paths. Figure 5 illustrates the timing faults in two FPGAs that run this same design with a 125ps phase difference between clk_l and clk_c. Initially, the entire capture register contains 0s (black color), and we launch the signal from the launch register of all the paths. During signal propagation from left to right, the capture register latches 1 (white color), but failures emerge at various points (red dots) due to insufficient time for the logic state to propagate the entire path. Despite identical design, these paths show distinct behaviors within the same device (*i.e.,* inter-path variation) and across different devices.

Figure 6a further highlights this variation through box-and-whisker plots that compare the iFFs of two devices under the same clock configuration. We perform a thousand launch-and-capture cycles for each device, revealing variability in timing failure due to operational factors like metastability. Although most paths behave consistently, outliers exist, necessitating consideration of operational variation in signature construction. Our experiments indicate that executing the fingerprint extraction process 100 consecutive times and using the *mode* of these measurements filters out instantaneous noise.

To demonstrate inter-FPGA variation in a large number of devices, we deploy a dynamic timing fault technique on 300 cloud FPGAs, extracting iFFs while maintaining the same phase relation between the clocks. Figure 6b shows a histogram of the failure indices for a path, revealing the variations among the FPGAs. In Figure 6c, we show the distribution of the average failure index of 32 paths in each device, demonstrating the inter-device dependency of crafted timing-based erroneous computations.

5.6 Parameter Selection

The experiments reveal three degrees of freedom to improve the amount of information in the fingerprinting approach. Firstly, the value stored in the launch register (1 or 0) effectively doubles the utility of a single path. Essentially, the path can propagate a 1 or a 0. The different values expose variation in different physical components in a CMOS gate: PMOS transistors control going from a 0 to a 1 logic level, while NMOS transistors control going from a 1 to a 0 logic level. Thus, for no additional hardware or time overhead, you can get double the number of paths. Secondly, increasing the number of paths increases the number of dimensions of comparison for the classifier. This increases hardware overhead, but is efficient from a measurement time perspective. Lastly, adjusting the phase relationship between clk_l and clk_c (i.e., T_S) allows a controlled exposure of process information at different points in a path (i.e., increase or decrease the number of transistors that whose variation contributes to each measurement). As depicted in Figure 7, extending the signal propagation time shifts the path timing fault toward higher indices, covering more elements in a physical path. The trade off of this tuning parameter is that too short a sample period causes devices to look similar as the signal does not propagate at all and too long a sample period suffers the averaging effect of the Law of Large Numbers and increases hardware overhead due to requiring more CARRY8s to capture the timing fault. We combine these three parameters

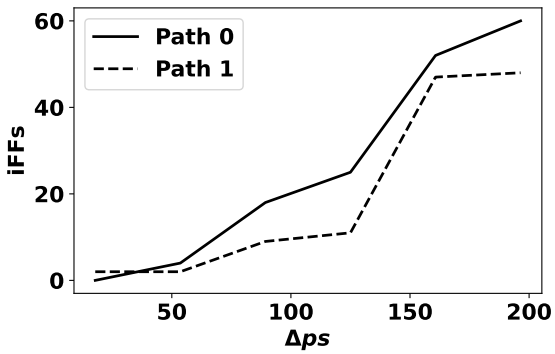


Figure 7. Dynamic control of the phase angle between clk_l and clk_c results in variations in the physical distance covered by different paths within the same FPGA. Although not clear for comparisons between individual paths, across many paths, paths tend to look more similar at low and high Δps.

to capture and construct an FPGA's signature, but leave a detailed exploration of the trade space to future work.

To capture the process variation, we tune the phase shift of an MMCM, optimizing the timing faults across all target paths. Our empirical analysis reveals that a VCO frequency of 1GHz, yielding a resolution of 17.8*ps*, combined with a 64-bit chain composed of 8xCARRY8 elements and 32 delay lines, achieves robust signature extraction and reliable FPGA identification across environmental noise and time.

6 Evaluation

This section details the empirical evaluation of *PhasePrint*, extending our preliminary experiments (§5). We examine the performance of *PhasePrint*, focusing on its accuracy, reliability, speed, and execution cost.

6.1 Experimental Setup

PhasePrint's design uses AWS hardware/software development kits and various `f1` instances. A central component in the development process is Amazon's FPGA Developer AMI [2], a specialized operating system image that provides a comprehensive development environment for Xilinx UltraScale+ FPGAs using the Vivado Design Suite, along with additional software frameworks. The FPGA boards connect via a PCIe bus and pair with an Intel(R) Xeon(R) Platinum 8259CL CPU to form a virtual machine with FPGA capabilities. These virtual machines come in three configurations: `f1.2xlarge`, `f1.4xlarge`, and `f1.16xlarge`, which include 1, 2, and 8 FPGA boards, respectively.[4]

To collect failure patterns, we launch all instances simultaneously to ensure no device overlap among allocation requests. AWS imposes a default limit of 160 virtual CPUs (vCPUs) for concurrently running instances per user. Increasing this limit is trivial, and we request AWS approval to raise our concurrent vCPU ceiling to 640, enabling us to run up to 10 `f1.16xlarge` instances[5] simultaneously in each region. As a user's allocation of unique FPGAs depends on regional availability, we extend our requests across additional geographic regions to augment our dataset.

We implement a fleet launching system, operated from our local workstation, that initiates requests for various `f1` instances across four AWS regions. This system deploys our custom Amazon Machine Image (§5) to these regions, maintaining consistent access key and security configurations to facilitate seamless data acquisition. Our launch system operates at the maximum capacity permitted by the increased limit, waits for the instances to be in a *running state*, and checks the signature flag that *PhasePrint*'s system service stores in an instance's file system to indicate whether it has completed extracting fingerprints of all the available FPGAs. When fingerprints are available, the launch system copies the data to our workstation for further processing and terminates all instances. Our dataset includes signatures from 300 different FPGAs. Table 1 lists the number of FPGAs per instance type and region.

6.2 Evaluating FPGA Detection Accuracy

We define an FPGA's fingerprint as a vector, with each element representing the specific location along a path where an unexpected value occurs following the launch instruction. That is, a vector of 32 dimensions where each dimension takes on a value from 0 to 64; mathematically expressed as a vector $\mathbf{v} = (v_1, v_2, \ldots, v_{32})$ where $v_i \in \{0, 1, 2, \ldots, 64\}$ for $i = 1, 2, \ldots, 32$.[6] This method of signature vectorization not only identifies the specific path of failure but also quantifies the extent of the failure within that path. Thus, considering the failures in a vector formation allows us to numerically capture the erroneous path *and* depth due to timing failure.

FPGA localization via fingerprint is a multi-class classification problem where each class denotes a unique device. We train support vector machine-based classifiers on fingerprint vectors for this task. As detailed in Section 5, we record each fingerprint 100 times to account for transient variations. This process results in 100 distinct vectors for each device, amassing a dataset of 30,000 data points. When we train the classifier with an 80/20 split between training and testing data for a specific Ts, it initially yields an accuracy of 91.41%. This accuracy improves when *PhasePrint* leverages

[4] AWS provides two types of instances: on-demand and spot. During development, we use low-cost spot instances (*i.e.*, `r5.2xlarge, t2.2xlarge`) to save costs. For all experiments, we switch to on-demand instances to ensure consistent availability and avoid interruptions.

[5] 80 f1.2xlarge, or 40 f1.4xlarge instances

[6] 64 represents a particular case when a logic state propagates to the end of the line for a given T_s.

Instances\Regions	N. Virginia us-east-1	Oregon us-west-2	Frankfurt eu-central-1	Ireland eu-west-1	# of FPGA/instance type
f1.2xlarge	0	0	2	20	22
f1.4xlarge	40	16	39	0	190
f1.16xlarge	0	4	0	7	88
# of FPGA/region	80	64	80	76	**300**

Table 1. List of FPGA instances in four AWS regions.

Features size	32	64	96
Accuracy	91.41%	98.97%	99.66%

Table 2. Allowing more time for signal propagation through physical paths improves classification accuracy.

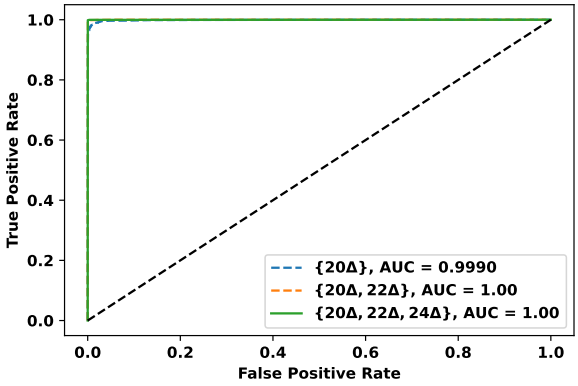
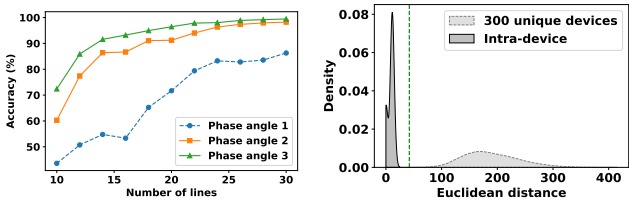


Figure 8. Micro-average ROC curves with corresponding AUCs showing the classifier's performance on a 300-FPGA dataset.



(a) **(b)**

Figure 9. (a) The graphs demonstrate that increasing the number of phase angles and paths provide us with more information about the underlying FPGA, leading to higher accuracy. (b) Intra-device fingerprint variation in 24 hours compared to inter-device variation.

additional phase angles in the fingerprint construction to cover more physical elements in a path (see Figure 7).

We add signature vectors by adjusting phase angles to cover longer physical paths and merge this additional data with vectors from previous phase angles. For example, vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ represent erroneous computation at two different phase angles between clk_l and clk_c. We combine them into a new vector $\mathbf{v_{12}} = \{\mathbf{v_1}, \mathbf{v_2}\}$, doubling the feature size from 32 to 64 and capturing the iFF for both phase angles. Incorporating data from three phase angles, **the classifier's accuracy reaches to 99.66%**, as shown in Table 2.

To better illustrate the impact of the number of phase angles and paths on the classification accuracy, we plot Figure 9a. As mentioned above, incorporating additional phase angles improves entropy, which ultimately increases classification accuracy.

We calculate a Receiver Operating Characteristic (ROC) curve and its area (AUC) to examine more nuanced performance metrics for the classification problem. Support vector-based models classify data across a boundary, which is a generally binary notion. As *PhasePrint* deals with a multi-class classification, we compare each FPGA's fingerprint with the rest of the FPGAs (*one-vs-rest* classification)— this leads to n classifiers classifying n number of FPGAs. We calculate ROC curve and AUC for these classes and create micro-averaged statistics to summarize the overall performance of the models.[7] Figure 8 shows each feature-size's micro-averaged ROC curve along the AUC value. As we add features by utilizing our dynamic phase tuning measure (§5), the classifier shows a near-perfect discriminatory ability to predict a specific device.

6.3 Fingerprint Robustness

A robust device signature must withstand various sources of variation, including temperature and power fluctuations. As demonstrated in Section 5, these transient variations have some influence on signal propagation (and timing failures). We filter out short-term fluctuations, such as those caused due to flip-flop metastability, by reading the signature 100 times and using the mode of these samples. However, we must also consider data center temperature and voltage variation in the longer term to assess whether a fingerprint remains distinguishable. Specifically, we investigate the stability of *PhasePrint*'s signature quality over an extended period, such as a day, and its effect on classification accuracy. We conduct a 24-hour experiment to assess this, extracting the signature from an FPGA every two hours.

Our experiment shows that most paths (27 out of 32) are robust against such variation, *i.e.,* their iFFs remain the same

[7]The area under the ROC curve (AUC) is a measure of the model's ability to distinguish between the classes [7]. An AUC of 1 indicates perfect classification, while an AUC of 0.5 suggests no discriminative power (gray dotted line in Figure 8), equivalent to random guessing.

across short-term (100 consecutive reads) and 24-hour periods. To examine the overall distribution of these fingerprints across 24 hours, we calculate a scalar metric— Euclidean distance[8]—of the fingerprints from its initial values. Then, we plot this device's distance distribution along with unique 300 FPGA's pairwise distribution [total $\binom{300}{2}$] in Figure 9b. As you can see, the transient variation remains far below the inter-device distances, which is also reflected in *the classification accuracy as it remains unchanged.*

Although the above experiment demonstrates robustness, users cannot control voltage and temperature variations in a data center. In high-noise environments, we can enhance classification performance by leveraging both temporal and spatial resources. Specifically, we increase the entropy of the FPGA fingerprint, defined as $entropy = f(\Delta, e, x)$, where Δ represents the phase angle, ranging from no propagation to full propagation; e denotes the signal edge (rising or falling), with different transistor sets handling each edge; and x refers to the number of paths and their respective lengths. The available CARRY8 elements in the FPGA limit the number of paths, while the smallest achievable phase angle constrains the minimum path length. We leave the task of exploring the relationships among these variables and optimizing entropy for future work.

6.4 Performance Comparison

Our work closely aligns with two notable previous works. The first is a DRAM-based method where AWS DRAM data retention is used to fingerprint DRAM and, hence, FPGA boards [38]. Adhering to AWS security protocols, this method proposes a high-accuracy cloud FPGA fingerprinting scheme by leveraging DRAM PUFs [22, 44, 45]. The DRAM-based approach begins by initializing the FPGA board's DRAM modules with a logic state using one image. It then leverages the f1 instance's data retention feature to maintain the DRAM contents across multiple images. A second image is subsequently loaded onto the FPGA; however, this image does not refresh the DRAM, causing some cells to lose their states. Addresses of the lost states vary across module, essentially creating DRAM PUFs [33]. Although this method is effective, it requires the sequential loading of three AFI images and a two-minute decay window to capture a DRAM-PUF response.

The second related work involves FPGA cartography via PCIe contention, analyzing the CPU-FPGA data transfer rate to identify co-located `f1` instances [35]. It shows how FPGA boards connect to a NUMA node by transferring 840MB data between CPU-FPGA through a shared PCIe bus, determining if an instance shares the same NUMA node (and thus the same physical CPU). This PCIe-based method *does not locate a specific FPGA* in the node.

[8]This is a straight line distance between two points in Euclidean space $||\mathbf{V_1}, \mathbf{V_2}|| = \sqrt{\sum_{i=1}^{96}(v1_i - v2_i)^2}$.

Launch pairs	launch {0,1}	launch {2,3}	launch {3,4}
Overlaps (%)	65%	56.25%	67.5%

Table 3. FPGA assignment overlaps between a pair of requests 10 minutes apart.

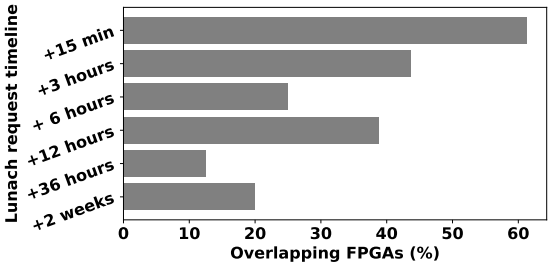


Figure 10. Overlaps between a set of 80 FPGAs and other launches across various time intervals.

In contrast, *PhasePrint* shifts the entire fingerprinting process on-chip, eliminating reliance on external components, and it needs only a single bitstream along with 765-byte data transfer between CPU and FPGA. The total execution time reduces to 9s, approximately 1/13 of the DRAM-based method. This efficiency gain enables the integration of our fingerprinting process into the FPGA instance's initialization phase, ensuring immediate availability of the signature upon user connection to the virtual machine. *PhasePrint*'s fast execution time allows rapid identification of FPGA instances when targeting a set of FPGAs in the cloud. The following section demonstrates why rapid execution is essential in a localization process.

6.5 AWS F1 Instance Placement Locality

PhasePrint uncovers the pattern of FPGA instance allocation by AWS during different launch events. By conducting repeated launch requests for 40 `f1.4xlarge` instances at varying intervals, we analyze the temporal locality in FPGA assignment. Specifically, we execute three sets of paired launch requests using our custom AMI integrated with *PhasePrint*'s hardware and software capabilities for fingerprint extraction. Each pair of requests is spaced about 10 minutes apart. Our findings indicate a significant overlap in FPGA assignments—over 50%—for temporally adjacent launches, as detailed in Table 3.

This overlapping persists even when launches are initiated from two distinct user accounts. The degree of overlap diminishes with increasing time intervals between launches. However, as depicted in Figure 10, a notable overlap in FPGA assignments is still evident, even for launches spaced weeks apart. Consequently, simply releasing and then re-renting FPGA sets to mitigate long-term data remanence risks is ineffective due to the high likelihood of overlap. Our finding reinforce the practicality of covert communication [37]

and side-channel attacks [9] among users renting FPGAs sequentially. *PhasePrint* offers a fast, cost-efficient solution for either launching these attacks or thwarting them by rapidly extracting and localizing FPGA fingerprints in a specific set of devices to avoid re-allocation of the same device.

6.6 Countermeasures

Given the offensive use cases for *PhasePrint*, it is important to provide guidance on potential mitigation strategies. Towards that direction, we qualitatively discuss potential mitigation strategies, but leave the building, evaluating, and circumventing of such systems as future work. From the high level, an effective countermeasure to *PhasePrint* must eliminate access to the analog-domain information that *PhasePrint*'s uses to identify cloud FPGAs.

We consider three mitigation strategies that embody two high-level directions: detection and prevention. Defenders could attempt to detect the presence of *PhasePrint* by performing exhaustive timing simulations. An exhaustive timing analysis, where tunable timing parameters are swept and the digital output of the target circuit is examined can reveal correlation between timing changes and timing faults (as they appear in the digital domain). Unfortunately, exhaustive timing analysis is both resources-intensive (e.g., time-consuming) and potentially ineffective, as no clear threshold defines when a phase relationship among clocks becomes a threat. Additionally, some circuitry's digital behavior is by-design correlated small changes in timing, e.g., high-speed serial and DRAM interfaces. Thus, an attacker could masquerade as or co-opt such trusted functionality to undermine timing analysis.

Given the challenges of detection, a defender may attempt to prevent *PhasePrint* entirely by eliminating access to its core building block: advanced clock generators. *PhasePrint* relies on advanced clock generators to create a small phase difference between the launch clock and the capture clock. Without the ability to create small phase differences, *PhasePrint* cannot capture analog-domain differences across cloud FPGAs, thus it cannot discriminate between cloud FPGAs. While this defense works, it works only for a subset of cloud uses cases that do not require direct access to high-speed serial communication (e.g., PCIe) or DRAM storage. It also imposes limitations on the clocks available to user designs as access to advance clock generators must be mediated by the cloud vendor. Even with such a defense in place, note that it is possible to re-implement *PhasePrint* using combinational logic as a source of phase shift, removing the need for advanced clock generation entirely. Using combinational logic as a source of phase shift is common in non-FPGA and earlier TDC designs.

Another prevention strategy is to introduce operational noise into the cloud FPGA deployment that can make delay measurements highly uncorrelated to analog-domain manufacturing-time variation. For example, injecting random noise into the cloud FPGA's power distribution network by updating power management firmware. While this step prevents precise timing measurements, it also requires balancing stable and efficient FPGA operation with security requirements. We imagine that attackers would respond to such a defense by collecting many measurements and leveraging techniques from Differential Power Analysis from the side-channel community to tease-out the FPGA specific component of variation.

7 Related Work

Integrating FPGAs into shared environments introduces various security challenges. This has led to extensive research efforts aimed at evaluating the security of cloud FPGAs, their host CPUs, and the broader cloud infrastructure [46].

Research shows that both temporal and spatial sharing of FPGAs lead to covert channels [11, 12, 29]. Temporal covert channels enable information transfer between temporally separated FPGA instances that share the same hardware. For instance, Tian *et al.* demonstrate circuit-generated heat in one FPGA instance transmits data to another instance upon reallocation on Microsoft Catapult servers [37]. A transmitting user deploys ROs [36] to increase an FPGA's temperature, while the receiving party uses the same circuit to decode the transmitted information. The assumption here is that both parties rent the same FPGA within a short time between release from the transmitting user and rent by the receiving user, which is possible as we have seen in the previous section. Concurrently running FPGA instances also establish covert communication by exploiting shared physical resources like power supplies and PCI buses in the same server. $C^3APSULe$ reveals how FPGA instances communicate covertly using power supply fluctuations when concurrently running on the same server [14]. Similarly, creating PCIe congestion using one FPGA and monitoring the bus from another enables covert communication if the transmitter and receiver are on the server [15]. High activity in one FPGA due to self-oscillating circuits can create enough voltage fluctuations that one can sense such variation from another FPGA sharing the same power supply unit. Similarly, creating a high-volume data transfer from the CPU to the FPGA creates congestion in a PCIe bus, which another FPGA senses by measuring latency in its data transfer using the same PCIe bus. In multi-tenant environments where multiple users share the same FPGA hardware, such covert communication becomes even more precise [13].

Similar to covert channels, many studies have investigated side channels among users in both temporal and spatial sharing environments. While academic community has shown considerable interest in spatial sharing setups, the current cloud business model limits FPGA access to one user at a time [37]. In the context of this paper, we focus on the

existing cloud setting where a single user occupies an entire FPGA. *Pentimento* is the first cross-user side channel to demonstrate that once an FPGA is reallocated to a new user, it leaks security-critical information from the previous user, thereby breaching the confidentiality assurances provided by cloud vendors [9]. *Pentimento* exploits the fact that FPGA fabric's analog property (*i.e.,* delay) deviates from its typical value depending on which logic state is stored for a long time. This system essentially retrieves a previous user's data from path delay deviations due to long-term data remanence. The effectiveness of such side channels largely depends on the attacker's ability to discern the prior analog characteristics of a path. *PhasePrint* addresses this challenge by providing a method to uniquely identify FPGAs, thereby closing the research gap and eliminating uncertainty about which path delay characteristics correspond to specific FPGAs.

FPGAs in the cloud environment bring new security challenges to the infrastructure itself [20, 25, 27]. For example, Gnad *et al.* reveal a voltage-based DoS attack on the FPGA servers [17]. Using self-oscillating circuits, they draw power beyond an FPGA's budget, rendering the device into a temporary unusable state. Tian *et al.* exploit PCIe congestion to map the cloud infrastructure and show how FPGA-CPU communication latency unveils the physical location of FPGA boards on the PCI bus [35].

8 Conclusion

We propose *PhasePrint*, a cloud FPGA fingerprinting system that leverages a commodity hardware design feature (Figure 2) to circumvent the stringent security measures enforced by cloud FPGA vendors. *PhasePrint* uses an FPGA's internal clock synthesizer to create a pair of clocks and dynamically adjusts their phase relation from software to trigger erroneous computation in a specially designed circuit. Theses timing errors cause manufacturing-induced variation to be revealed in the digital domain, acting as device identifiers. Our evaluation of 300 FPGAs on AWS demonstrates a 99.7% accuracy, achieving 13× faster performance and reduced costs compared to previous methods.

Security of the FPGAs in the cloud is a growing concern due to the novel sytem security threats created by custom hardware on shared devices. A key assumption underpinning these attack models is the capability to pinpoint a specific FPGA in the cloud. This paper removes this assumption by introducing a method that one can leverage for both facilitating and thwarting this new class of attacks. The use of generic design components such as common circuit elements and timing manipulation methods in *PhasePrint* highlights a critical need for cloud vendors to fundamentally rethink their defensive strategies. We posit that as long is it is possible for cloud hardware to have user-exposed timing errors, it is possible to differentiate that hardware in the cloud.

Acknowledgment

We thank the anonymous reviewers for their feedback and our shepherd Dustin Richmond for his guidance on the final version of this paper. The project depicted is sponsored by the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited. This material is based upon work supported by the National Science Foundation under Grant No. 2046589.

References

[1] AWS. 2023. *AFI Power.* https://github.com/aws/aws-fpga/blob/master/hdk/docs/afi_power.md

[2] AWS. 2023. *Amazon FPGA developer AMI.* AWS marketplace. https://aws.amazon.com/marketplace/pp/prodview-gimv3gqbpe57k

[3] AWS. 2023. AWS FPGA Management Library. https://github.com/aws/aws-fpga/tree/master/sdk/userspace/fpga_libs/fpga_mgmt.

[4] AWS. 2023. *AWS FPGA PCIe Memory Map.* https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Fpga_Pcie_Memory_Map.md

[5] AWS. 2023. *AWS Shell Interface Specification.* https://github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md

[6] AWS. 2023. *Using Data Retention to Preserve the Contents of DRAM Across AFI Loads.* https://github.com/aws/aws-fpga/blob/master/hdk/docs/data_retention.md

[7] Christopher M Bishop. 2006. *Pattern Recognition and Machine Learning by Christopher M. Bishop.* Springer Science+ Business Media, LLC.

[8] Colin Drewes, Tyler Sheaves, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. 2024. Turn on, Tune in, and Listen up: Maximizing Side-Channel Recovery in Cross-Platform Time-to-Digital Converters. *ACM Transactions on Reconfigurable Technology and Systems* 17, 3 (2024), 1–30.

[9] Colin Drewes, Olivia Weng, Andres Meza, Alric Althoff, David Kohlbrenner, Ryan Kastner, and Dustin Richmond. 2024. Pentimento: Data remanence in cloud FPGAs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 862–878.

[10] Colin Drewes, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. 2023. Turn on, tune in, listen up: Maximizing side-channel recovery in time-to-digital converters. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 111–122.

[11] Ilias Giechaskiel, Ken Eguro, and Kasper B Rasmussen. 2019. Leakier wires: Exploiting FPGA Long Wires for Covert-and Aide-channel Attacks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12, 3 (2019), 1–29.

[12] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Ken Eguro. 2016. Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2016). https://doi.org/10.1145/3196494.3196518

[13] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Jakub Szefer. 2019. Reading Between the Dies: Cross-SLR Covert Channels on Multi-Tenant Cloud FPGAs. *2019 IEEE 37th International Conference on Computer Design (ICCD)* (2019), 1–10. https://doi.org/10.1109/ICCD46524.2019.00010

[14] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Jakub Szefer. 2020. C3APSULe: Cross-FPGA Covert-Channel Attacks through Power Supply Unit Leakage. *2020 IEEE Symposium on Security and Privacy (SP)*

(2020), 1728–1741. https://doi.org/10.1109/SP40000.2020.00070

[15] Ilias Giechaskiel, Shanquan Tian, and Jakub Szefer. 2021. Cross-VM Information Leaks in FPGA-Accelerated Cloud Environments. *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* (2021), 91–101. https://doi.org/10.1109/HOST49136.2021.9702277

[16] Ognjen Glamočanin, Louis Coulon, Francesco Regazzoni, and Mirjana Stojilović. 2020. Are Cloud FPGAs Really Vulnerable to Power Analysis Attacks?. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1007–1010.

[17] Dennis RE Gnad, Fabian Oboril, and Mehdi B Tahoori. 2017. Voltage Drop-based Fault Attacks on FPGAs Using Valid Bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

[18] Dennis R. E. Gnad, C. Nguyen, Syed Hashim Gillani, and M. Tahoori. 2021. Voltage-Based Covert Channels Using FPGAs. *ACM Trans. Design Autom. Electr. Syst.* 26 (2021), 43:1–43:25. https://doi.org/10.1145/3460229

[19] Dennis R. E. Gnad, Sascha Rapp, Jonas Krautter, and Mehdi B. Tahoori. 2018. Checking for Electrical Level Security Threats in Bitstreams for Multi-tenant FPGAs. In *2018 International Conference on Field-Programmable Technology (FPT)*. 286–289. https://doi.org/10.1109/FPT.2018.00055

[20] Mathieu Gross, Jonas Krautter, Dennis Gnad, Michael Gruber, Georg Sigl, and Mehdi Tahoori. 2023. FPGANeedle: Precise Remote Fault Attacks from FPGA to CPU. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. 358–364.

[21] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with {AmorphOS}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 107–127.

[22] Jeremie S Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. 2018. The DRAM latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 194–207.

[23] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. 2018. FPGAhammer: Remote voltage fault attacks on shared FPGAs, suitable for DFA on AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 44–68.

[24] T. La, Kaspar Matas, Nikola Grunchevski, K. Pham, and Dirk Koch. 2020. FPGADefender: Malicious Self-oscillator Scanning for Xilinx UltraScale + FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 13 (2020), 15:1–15:31. https://doi.org/10.1145/3402937

[25] Tuan La, Khoa Pham, Joseph Powell, and Dirk Koch. 2021. Denial-of-service on FPGA-based cloud infrastructures—attack and defense. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 441–464.

[26] Jubayer Mahmod and Matthew Hicks. 2024. *PhasePrint Source Code Repository*. https://github.com/FoRTE-Research/PhasePrint

[27] Dina G Mahmoud, David Dervishi, Samah Hussein, Vincent Lenders, and Mirjana Stojilović. 2022. DFAulted: Analyzing and Exploiting CPU Software Faults Caused by FPGA-Driven Undervolting Attacks. *IEEE Access* 10 (2022).

[28] Mehrdad Majzoobi and Farinaz Koushanfar. 2011. Time-bounded authentication of FPGAs. *IEEE Transactions on Information Forensics and Security* 6, 3 (2011).

[29] S. Mirzargar and Mirjana Stojilović. 2019. Physical Side-Channel Attacks and Covert Communication on FPGAs: A Survey. *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (2019), 202–210. https://doi.org/10.1109/FPL.2019.00039

[30] Michael Moukarzel and Matthew Hicks. 2021. RingRAM: A Unified Hardware SecurityPrimitive for IoT Devices that Gets Better with Age. In *Annual Computer Security Applications Conference*. 660–674.

[31] George Provelengios, Daniel Holcomb, and Russell Tessier. 2019. Characterizing power distribution attacks in multi-user FPGA environments. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 194–201.

[32] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. 199–212.

[33] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Boris Škorić, Stefan Katzenbeisser, and Jakub Szefer. 2018. Decay-based DRAM PUFs in commodity devices. *IEEE Transactions on Dependable and Secure Computing* 16, 3 (2018), 462–475.

[34] David Spielmann, Ognjen Glamočanin, and Mirjana Stojilović. 2023. RDS: FPGA Routing Delay Sensors for Effective Remote Power Analysis Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023, 2 (2023), 543–567.

[35] Shanquan Tian, Ilias Giechaskiel, Wenjie Xiong, and Jakub Szefer. 2021. Cloud FPGA Cartography using PCIe Contention. *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2021), 224–232. https://doi.org/10.1109/FCCM51124.2021.00035

[36] Shanquan Tian, Andrew Krzywosz, Ilias Giechaskiel, and Jakub Szefer. 2020. Cloud FPGA Security with RO-based Primitives. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 154–158.

[37] Shanquan Tian and Jakub Szefer. 2019. Temporal Thermal Covert Channels in Cloud FPGAs. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2019). https://doi.org/10.1145/3289602.3293920

[38] Shanquan Tian, Wenjie Xiong, Ilias Giechaskiel, K. Rasmussen, and Jakub Szefer. 2020. Fingerprinting Cloud FPGA Infrastructures. *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2020). https://doi.org/10.1145/3373087.3375322

[39] Neil HE Weste and David Harris. 2015. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India.

[40] Wikipedia. 2023. *Unary Coding*. https://en.wikipedia.org/wiki/Unary_coding#:~:text=Unary%20coding%2C%20or%20the%20unary,followed%20by%20a%20zero%20(if

[41] Xilinx. 2019. *MMCM and PLL Dynamic Reconfiguration*. https://docs.xilinx.com/v/u/en-US/xapp888_7Series_DynamicRecon

[42] Xilinx. 2021. *UltraScale Architecture Libraries Guide*. https://docs.xilinx.com/viewer/book-attachment/1y6wgYUj4VyKLeZeHJRFfg/rLt3BZCWpfCQmfRbdX4Nig Accessed: 2023.

[43] Xilinx. 2022. *UltraScale Architecture Clocking Resources User Guide*. https://www.xilinx.com/support/documents/user_guides/ug572-ultrascale-clocking.pdf Accessed: 2023.

[44] Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. 2019. Spying on Temperature using DRAM. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 13–18. https://doi.org/10.23919/DATE.2019.8714882

[45] Wenjie Xiong, André Schaller, Nikolaos A Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. 2016. Run-time Accessible DRAM PUFs in Commodity Devices. In *18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings 18*. Springer, 432–453.

[46] Shaza Zeitouni, Ghada Dessouky, and Ahmad-Reza Sadeghi. 2020. SoK: On the Security Challenges and Risks of Multi-tenant FPGAs in the Cloud. *arXiv preprint arXiv:2009.13914* (2020).

[47] Mark Zhao and G Edward Suh. 2018. FPGA-based Remote Power Side-channel Attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 229–244.