information

Article

FEINT: Automated Framework for Efficient INsertion of Templates/Trojans into FPGAs

Virinchi Roy Surabhi [1], Rajat Sadhukhan [1], Md Raz [1], Hammond Pearce [2], Prashanth Krishnamurthy [1,*], Joshua Trujillo [3], Ramesh Karri [1] and Farshad Khorrami [1]

[1] Department of Electrical and Computer Engineering, New York University Tandon School of Engineering, Brooklyn, NY 11201, USA; virinchi.roy@nyu.edu (V.R.S.); rs9087@nyu.edu (R.S.); md.raz@nyu.edu (M.R.); rkarri@nyu.edu (R.K.); khorrami@nyu.edu (F.K.)
[2] School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia; hammond.pearce@unsw.edu.au
[3] Department of Energy's Kansas City National Security Campus, Kansas City, MO 64147, USA; jtrujillo@kcnsc.doe.gov
* Correspondence: prashanth.krishnamurthy@nyu.edu

Abstract: Field-Programmable Gate Arrays (FPGAs) play a significant and evolving role in various industries and applications in the current technological landscape. They are widely known for their flexibility, rapid prototyping, reconfigurability, and design development features. FPGA designs are often constructed as compositions of interconnected modules that implement the various features/functionalities required in an application. This work develops a novel tool FEINT, which facilitates this module composition process and automates the design-level modifications required when introducing new modules into an existing design. The proposed methodology is architected as a "template" insertion tool that operates based on a user-provided configuration script to introduce dynamic design features as plugins at different stages of the FPGA design process to facilitate rapid prototyping, composition-based design evolution, and system customization. FEINT can be useful in applications where designers need to tailor system behavior without requiring expert FPGA programming skills or significant manual effort. For example, FEINT can help insert defensive monitoring, adversarial Trojan, and plugin-based functionality enhancement features. FEINT is scalable, future-proof, and cross-platform without a dependence on vendor-specific file formats, thus ensuring compatibility with FPGA families and tool versions and being integrable with commercial tools. To assess FEINT's effectiveness, our tests covered the injection of various types of templates/modules into FPGA designs. For example, in the Trojan insertion context, our tests consider diverse Trojan behaviors and triggers, including key leakage and denial of service Trojans. We evaluated FEINT's applicability to complex designs by creating an FPGA design that features a MicroBlaze soft-core processor connected to an AES-accelerator via an AXI-bus interface. FEINT can successfully and efficiently insert various templates into this design at different FPGA design stages.

Keywords: FPGA; template insertion; EDA; place-and-route; Trojan insertion

Citation: Surabhi, V.R.; Sadhukhan, R.; Raz, M.; Pearce, H.; Krishnamurthy, P.; Trujillo, J.; Karri, R.; Khorrami, F. FEINT: Automated Framework for Efficient INsertion of Templates/Trojans into FPGAs. *Information* **2024**, *15*, 395. https://doi.org/10.3390/info15070395

Academic Editor: Paolo Maistri

Received: 20 May 2024
Revised: 22 June 2024
Accepted: 26 June 2024
Published: 8 July 2024

1. Introduction

Field-Programmable Gate Arrays (FPGAs) are rapidly gaining popularity as a flexible hardware platform for prototyping and deploying various applications [1]. Product development with FPGAs is an efficient and cost-effective approach as they offer a broad selection of intellectual property (IP) modules for easy integration and reconfiguration [2]. These key benefits have resulted in a significant rise in the integration of FPGAs across various domains, including data center applications [3], wireless connectivity, the acceleration of artificial intelligence (AI) and machine learning (ML) tasks [4], IoT/embedded devices, edge computing applications, as well as cyber-physical systems, among others. Therefore, with FPGAs being widely used in safety-critical applications, there is a heightened need

to prioritize security and privacy measures while also optimizing their performance concerning power, performance, and area metrics. One significant hurdle when leveraging the considerable performance benefits of FPGAs is their programmability. Programming FPGAs is commonly viewed as a practice centered around the development of control paths, data paths, and finite-state machine design; thus, it requires a high level of hardware expertise. If the algorithm or functionality of an FPGA is supposed to accelerate changes or add-ons frequently, it can lead to obsolescence issues. FPGA designs do not readily accommodate algorithm modifications or additions as seamlessly as software designs, which can be easily recompiled or updated. Additionally, FPGA programming often relies on vendor-specific development tools and ecosystems, which can be less standardized and user-friendly compared to general-purpose programming environments. Hence, there is an imperative requirement to have a general platform-independent framework that can be used for adding functionalities to an FPGA design at various stages of its development using an interface that is defined by the design [5].

The development of products based on FPGAs involves various stages and interactions with third parties. These untrusted sources have the potential to maliciously modify a hardware IP block that is programmed onto an FPGA device at different stages of the FPGA life cycle [6]. These malicious alterations have attracted significant research attention, which has been focused on identifying and preventing them. Additionally, a plethora of hardware-based countermeasures exist to thwart these attacks either by inserting them manually in the design or through some platform-specific EDA flow. Some of these malicious alterations and countermeasures are as follows:

- Hardware Trojan Insertion [7]: Trojan circuits are designed and inserted in FPGAs with the intent to illicitly alter the FPGA behavior. They trigger system malfunctions, enable unauthorized remote access to hardware components, and monitor or leak sensitive data. A plethora of tools based on ML [8], aging [9], or exploring signal correlation and cyclic redundancy checks [10] exist in the literature to detect hardware Trojans in FPGAs.
- Side-channel and Fault Injection Attacks [11–13]: Side-channel attacks exploit the physical information that becomes apparent when a system employs an encryption technique to break into an FPGA. For fault attacks, adversaries inject faults into the behavior and then study the faulty behavior to retrieve secrets. Prominent hardware countermeasures [14–16] are inserted during FPGA development.
- Thermal Laser Simulation [17]: Using a current monitoring laser stimulation, the device is biased with a supply voltage, and the current between the supply pins is monitored via current pre-amplifier to retrieve the AES secret-key. These authors also proposed noise-based hardware mitigation [17].
- Reverse Engineering [18]: The adversary intercepts the generated bitstream to employ reverse-engineering techniques for retrieving higher-level functionality or structure-level descriptions. The countermeasure scheme involves obfuscating [19,20] the bitstreams to protect the IPs from typical reverse engineering attacks.
- Boolean Satisfiability(SAT)-Attacks [21]: This approach permits an attacker to decode an encrypted netlist by employing a set of meticulously chosen input patterns along with their output observations. A number of locking techniques [22,23], designed by tweaking the circuits, have been proposed to thwart this category of attacks.

However, all the above attack/mitigation tools and techniques have been developed and tested using different FPGA platforms and families. Hence, platform-dependent solutions do not promote interoperability between different design tools and platforms. Additionally, platform-dependent solutions cannot be adapted to new technologies and platforms, and they are not pluggable to new methods. Hence, there is an imperative need for a platform-independent integrated automated framework that can act as a general-purpose function inserter, can augment mitigation techniques against hardware attacks, or is able to insert Trojans to examine its effect. The framework should ideally possess the following desirable characteristics:

1. Serve as a versatile solution to augment design functions using a flexible and scalable interface defined by the design.
2. Support *a module composition-based design* that envisions a library of modules for error checking, monitoring, etc., which defenders can integrate into their designs to mitigate and thwart varied attack dimensions on FPGAs.
3. Facilitate the investigation of hardware attacks in FPGA netlists, seamlessly incorporating various categories of FPGA-specific Trojans into a netlist at various stages of the FPGA design cycle, thus enabling a swift and thorough examination of potential Trojan attacks.
4. Support automated cross-platform application by not being dependent on vendor-specific file formats and operating instead on industry-standard text-based formats that are compatible with different FPGA families, thus reducing need for manual programming and specialized hardware expertise.

In this work, we propose an automated framework FEINT, as shown in Figure 1, to encompass these features. FEINT can be used by an attacker to explore the domain of hardware Trojan attacks within FPGA netlists, or by a defender to insert hardware countermeasures to mitigate against various physical attacks in FPGAs. The proposed framework is useful in general for an amateur designer to insert add-on functionalities to an FPGA design by tuning the configuration files.

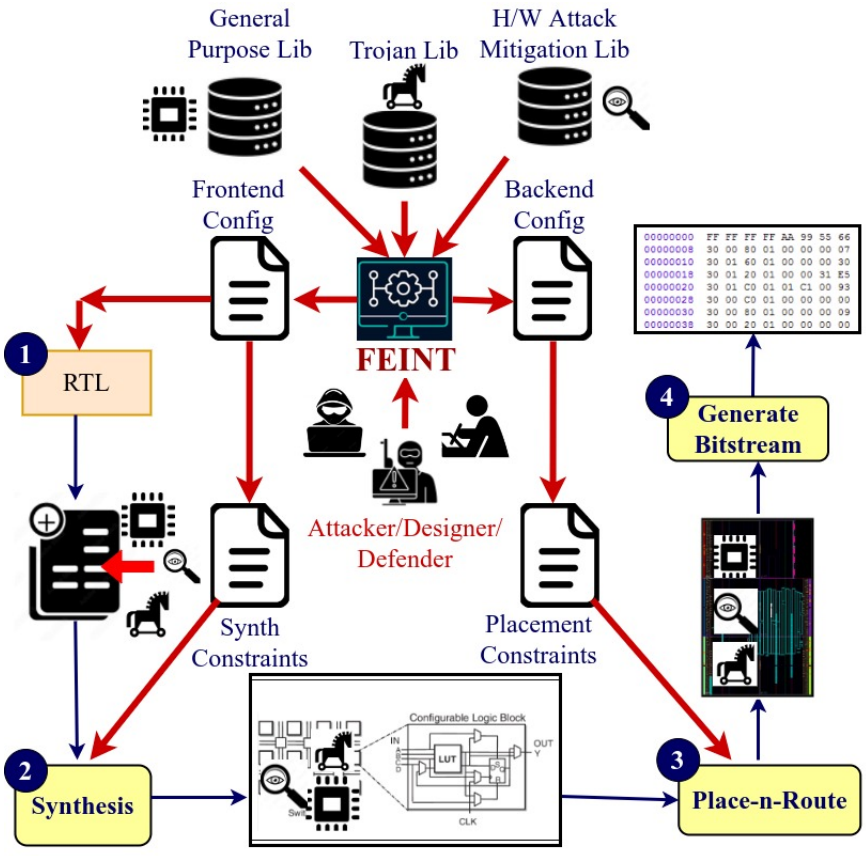

Figure 1. FEINT Flowcharts showing designer's, defender's, and attacker's perspectives.

The FEINT framework is illustrated in Figure 1, and it can be used by individuals in the roles of an attacker, a defender, or a designer. The attacker, defender, or designer employs a pre-defined Trojan library, hardware attack mitigation library, or general-purpose template library, respectively, to integrate modules either during the (1) RTL stage, (2) synthesis stage, or (3) the place-and-route stage within the FPGA design process. The general-purpose template library may consist of sensor modules, wireless modules, etc. The Trojan

library consists of various hardware Trojans. The attack mitigation library may have modules such as a power-on self-test, threshold implementation, and Hash algorithms. The FEINT approach merely requires script configuration to produce the desired hardware, thus ensuring independence from vendor-specific formats and platforms.

While FEINT can be applied for various purposes, including adding security features, improving functionality, or inserting hardware Trojans, we would like to note that FEINT by itself does not pose a security issue in terms of malicious use. This is because FEINT is simply an FPGA design modification tool (i.e., a general-purpose tool like a compiler) and security measures to prevent malicious use effectively entails ensuring that the computer/network infrastructure in which the FPGA design to be protected is located is secured against attackers. In terms of a threat model, FEINT does not increase the attack surface since any malicious use achievable using FEINT could be performed "by hand" by an adversary who gains access to a secure computer/network—FEINT simply reduces the manual workload to effect modifications of the FPGA code. These capabilities, therefore, enable a flexible tool that can be effectively used for multiple purposes, as discussed in this paper.

The rest of this paper is organized as follows: In Section 2, we provide an overview of the related works in this domain and elaborate our key contributions in this paper. Section 3 describes the tool capabilities, and Section 4 demonstrates the efficiency of our flow through the experimental results. Finally, Section 5 concludes the workm along with a brief note on future directions.

2. Related Works

In this section, we compare our proposed tool FEINT with the state-of-the-art EDA tools in this domain. A summary of related works, along with a comparison to our work, is shown in Table 1.

Table 1. Comparison of FEINT with state-of-the-art tools (✕✕—NA, ✔✕—applicable at RTL netlist stage, ✕✔—applicable at post-placement netlist stage, and ✔✔—applicable at RTL netlist and post-placement netlist stages).

Flow	Template Insertion		
	Generic Module	**H/W Attack Mitigation**	**Trojan Insertion**
RapidWright [24]	✕✔	✕✕	✕✕
R. Backasch [25]	✕✔	✕✕	✕✕
Athanas et al. [26]	✔✕	✕✕	✕✕
J Cruz et al. [27]	✕✕	✕✕	✕✔
HAL [28]	✕✕	✔✔	✕✕
TAINT [29]	✕✕	✕✕	✔✔
This work	✔✔	✔✔	✔✔

We conducted a comprehensive analysis of our approach in relation to other published flows across three aspects of template insertion, namely generic module insertion, hardware-based template insertion for safeguarding against physical attacks, and Trojan insertion. In our comparative assessment, we evaluated the adaptability of our proposed tool by considering both RTL netlist and post-placement netlist template insertion in comparison to other tools.

Our Contributions: FEINT provides a versatile design-enhancement interface and supports module composition-based design for error checking and monitoring, enabling defense against diverse FPGA attack dimensions. Additionally, it investigates hardware Trojan attacks, seamlessly integrating FPGA-specific Trojans at various design stages for analysis. The main contributions are as follows:

- FEINT is a flexible, versatile, and platform-independent Python tool chain for inserting design templates through an interface determined by the design. FEINT can be used by three categories of users, namely an adversary to insert or analyze the effect of Trojans, a designer to plug-in templates, and a defender to insert countermeasures to the design to protect against physical attacks.
- As a proof of concept, we created three template libraries: a general-purpose library, a Trojan library, and an attack detection/mitigation library. We then used FEINT to insert these user-chosen templates in a given user-provided design during the front or back end of the FPGA design flow.
- We validated the efficiency of our flow from three different perspective users, namely attacker, defender, and designer, using the Xilinx platform. We inserted 5 different Trojans to serve as part of the attacker's perspective, 7 out 15 tests were from a NIST randomness test suite as part of the defender's perspective, and we inserted 4 pseudo-random number generators into an existing design as part of the designer's perspective. We validated our modified designs using the CMOD-A7 (Artix) FPGA board.

In the next section, we will delve deeper into the front end and back end capabilities of our FEINT flow.

3. FEINT: Tool Capabilities

The automated template insertion tool is intended for use in digital design to alter the behavior of digital circuits. It is designed to read, modify, and write RTL/netlist hardware description files. It takes user-defined parameters such as signal names, module names, and filenames to perform the necessary modifications. The core functionality of the tool involves renaming modules, adjusting signal declarations, and managing signal connections. The tool is implemented as a Python script and operates on industry-standard, text-based formats for RTL/netlist; thus, it is cross-platform and not dependent on vendor-specific FPGA families and related FPGA synthesis tools. Our tool operates on industry-standard RTL/netlist text files and can be applied across FPGA families and design tool suites (and their respective versions). We define the changes made to the Verilog files, such as RTL level and netlist level files, as front end capabilities, and the changes made to the constraints such as location and region as back end capabilities. We elaborate on the two capabilities in the following sections, and we study the application of the tool in attacker, defender, and designer contexts. Through the modular structure of FEINT with an automated workflow in both front end and back end capabilities, FEINT enables rapid composition-based design and system customization with reduced manual effort.

3.1. Front End Capablities

3.1.1. Module Renaming

This tool generates a new module name by selecting N (=10 by default) random letters from the alphabet. This new name is then used to replace occurrences of the original module name within the RTL/netlist file. This functionality allows the tool to rename the module while maintaining the overall structure of the design.

3.1.2. Signal Declarations and Connections

FEINT's capability to add, modify, or remove signals can be used to introduce new signals or modify existing signals to contain malicious logic. The tool's manipulation of signal connections can be exploited to establish communication pathways between the inserted module and external entities. FEINT handles various types of signal modifications:

- Input and Output Signals: This tool adds input and output signals to the module declaration based on user-provided parameters. It accounts for cases where certain signals need to be omitted.

- Route Signals: Route signals are introduced to connect different parts of the design. The tool creates wire signals and manages the connections between these route signals and the original output signals.
- Join Signals: Join signals are used to combine or join existing signals. The tool replaces the original signal name with the join signal name, thus allowing the design to incorporate these combined signals.
- Extra Signals: Additional signals specified in the extra_signals parameter are seamlessly integrated into the module declaration.

3.1.3. File Manipulation

FEINT reads the RTL netlist, applies modifications, and writes the modified RTL netlist to the file. Overall, FEINT provides a valuable functionality for modifying HDL files in digital designs. It can rename modules, conduct signal declaration and connection management, and perform RTL netlist manipulation. By utilizing FEINT, designers can customize and adapt the FPGA designs to meet requirements, integrate subsystems, and streamline design processes. FEINT's features provide attackers with the ability to obfuscate and implant malicious logic within the FPGA designs. FEINT also facilitates defender evaluations of the vulnerabilities to Trojans in FPGA designs and the study of Trojan impacts.

3.2. Back End Capabilities

The main back end capability explored was the ability of the automated tool to place inserted designs within a desired area or location, as circuit placement during insertion can have a critical effect on the power, performance, and area parameters of the overall design. Several cross-platform approaches (portable to different FPGA vendors and families) to placement constraints were considered, each with different outcomes and varying impact depending on the placement objective. They included a location constraint, a region constraint, and a hybrid approach, utilizing both of the aforementioned constraint types.

3.2.1. Location Constraints

This type of constraint (Listing 1) allows for the placement of any design instance, such as a Look Up Table (LUT) or multiplexer (Mux), to a designated FPGA fabric instance of the same type. Between EDA vendors, these constraints may vary in granularity, i.e., constraining a cell to a specific slice vs. constraining a cell to a specific LUT within a slice, but they employ similar command structures and invoke the same outcome in placement.

Listing 1. Location Constraint Pseudo-command.

```
1    assign_location <Location> <Instance>
2
```

The command structure, depicted in the pseudo-command in Listing 1, generally involves a function `assign_location`, which evokes a location constraint and sets the granularity along with the design cell instance name `<Instance>` and the desired fabric location `<Location>` to be set during the place-and-route portion of the design flow. These location constraint commands can be generally placed within a constraints file, applied graphically via a floor-planning tool, or invoked through a command-line interface (CLI) using a Tool Command Language (TCL). Instances that are constrained using this approach are known to be "fixed," meaning they will not be relocated during optimizations, regardless of negative slack values or design timing closure status. Due to this, an attacker or designer must take care in determining which location designations are viable in meeting all design requirements. Furthermore, cross-platform, industry-standard synthesis attributes can be utilized within the RTL front end to aid in the automation and scripting of location constraints—they can be used to generate cell names in a predictable and consistent pattern, or prevent certain modules from being altered through optimizations. This approach has the tendency to re-route and relocate parts of the original design in order to meet timing and area constraints. This can be beneficial when a designer wants to achieve timing closure,

but can be detrimental when an attacker wants to conceal a Trojan with the least impact to the original design.

3.2.2. Region Constraints

This type of constraint (Listing 2) allows for the placement of large portions of logic within a set rectangular region. This allows for entire modules or design subsections to placed together within the same region. Furthermore, this approach allows for the segmentation of the original design and newly inserted Trojan or template.

Listing 2. Region Constraint Pseudo-command.

```
create_region <Name>
assign_logic <Name> <Inst_1 ... Inst_N>
size_region <Name> <Loc_1 Loc_2>

```

The portable cross-platform command structure of generating a region constraint, as shown in Listing 2 using pseudo-commands, usually involves three or more parts. The first part, `create_region`, creates a new region named `<Name>`. After the creation of the region, logic is assigned to this region using the `assign_logic` pseduo-command, with the arguments of the command including the name of the region and cell instances, `Inst_1 ... Inst_N`.

After cells are defined within the region, the shape and size of the region is defined using the `size_region` pseudo-command, with arguments being the region name along with a rectangle of two or more points, which are defined by fabric location instances, `<Loc_1 Loc_2>`. Region constraints, similar to the location constraints, can be set in a constraints file, applied graphically via a floor-planning tool, or invoked through a CLI using TCL commands, thus allowing for a variety of automation and scripting opportunities.

Unlike a location constraint, however, region constraints are not "fixed", meaning that the logic within them are free to be placed and routed based on efficiency and optimization. If the encompassing region is too small, does not have the required resources (such as block RAM), or if there are no viable routing/placement locations, elements may be placed outside the boundary rectangle. Concentric region constraints can be used to place the newly inserted Trojan or template anywhere within the original circuit while keeping the original circuit in the same area.

3.2.3. Hybrid Approach

This approach uses both the location constraint and region constraint in tandem to constrain the original circuit to a designated area while allowing newly inserted templates to be placed anywhere within the FPGA fabric. Since both fixed and unfixed placement constraints are used, care must be taken to ensure the fixed placement of newly added circuitry does not violate any timing constraints, especially if the newly added circuit exists within the design hierarchy of the original circuit. Overall, these approaches can be prescribed based on the objective of the attacker or designer, whether it be efficiency or concealment. The results, which demonstrate the application of these placement constraints concerning Trojan insertion when using the Xilinx Vivado toolchain and design flow, elucidate these considerations. We will elaborate on the experimental results in more depth in the next section.

4. Experimental Validation

In this section, we present a series of case studies aimed at comprehensively evaluating the potential applications of the FEINT tool. These case studies adopt three distinct perspectives: the attacker's, the defender's, and the designer's. Each perspective serves as a lens through which we assess the multifaceted utility of the tool. The first dimension of our case studies involves an examination of the tool from an attacker's standpoint. We simulate malicious intent by employing the tool to insert diverse types of Trojans into pre-existing designs. From the defender's perspective, we investigate the application of the tool to enhance design integrity and security. Our focus here is on the insertion of a Built-In Self-Test (BIST) architecture using the tool. The third dimension of our exploration

pertains to the designer's viewpoint. We used the tool to simplify integrating common HDL modules into the existing designs.

4.1. Experimental Setup

The testbed that was utilized experimentally when performing each of the three case studies included the use of a popular FPGA on a widely available hardware development board and a toolchain capable of accepting an industry-standard HDL. The Xilinx Artix-35T FPGA on the CMOD-A7 development board manufactured by Digilent was used to fit these criteria. This FPGA and board combination allowed for the use of the Vivado toolchain and design flow, which allows for the synthesis and implementation of RTL files. Furthermore, Vivado employs the use of synthesis attributes and constraint declarations similar to that of other vendors, thus allowing the FEINT tool to take advantage of these features with little to no modifications [30]. The same hardware setup was used across all three use case scenarios.

4.2. Front End Application

4.2.1. Attacker's Perspective

To illustrate the utilization of the FEINT tool from an attacker's standpoint, we employed it to insert hardware Trojans into an existing AES core design. In all the experiments in this section, we used the secret key 0x12233445 56677889 9AABBCCD DEEFF001. This investigation aims to showcase the tool's capabilities through the injection of five distinct Trojans, each featuring different attack vectors, as delineated in Table 2. For instance, in Figure 2, the top demonstrates the successful injection of circuitry for the Trojan labeled as LEAK_OUT:STATE using the tool. This Trojan leaks the secret key through the output bus upon activation. The block diagram of this Trojan is shown in Figure 2 in the middle section. The signals clock, key_in, resetn, and text_in are joined with the corresponding signals of the Trojan module. The signal text_out is rerouted through the Trojan module. The trigger of this Trojan operates by a single comparison operation. Figure 2 in the middle section demonstrates the Vivado schematic of this comparison mechanism. Figure 2 in the bottom section shows the implemented layout where the inserted Trojan is highlighted in white. Each value of the plain text is compared against the value 0x00112233 44556677 8899AABB CCDDEEFF. Notably, the tool empowers precise Trojan placement, a feature specified through its back end capability.

Table 2. Trojan templates inserted using the FEINT tool.

Trojan	Effect
LEAK_OUT:STATE	Leaks secret key via output using single-input trigger
LEAK_OUT:COUNTER	Leaks secret key via output using counter value trigger
LEAK_OUT:SEQ	Leaks secret key via output using four sequential triggers
LEAK_SIDE:STATE	Leaks secret key via side-channel using single-input trigger
DOS_OUT:SEQ	Performs Denial-of-Service (DoS) using four sequential triggers

The insertion of LEAK_OUT:COUNTER is shown in Figure 3 in the top section. The signals and their manipulations that were involved while inserting LEAK_OUT:COUNTER were the same as that of LEAK_OUT:STATE. This Trojan uses a two-bit counter to determine the status of Tj_Trig (Trojan trigger). The functionality of this counter is shown in Figure 3 in the bottom section. The counter is assigned a value of 0x0 at reset. Every time an encryption is completed, the counter is incremented. When the counter has a value of 0x3, the circuit is triggered. The functionality of LEAK_OUT:COUNTER is the same as that of LEAK_OUT:STATE. When the Trojan has been triggered, the cipher output will be replaced with the provided key.

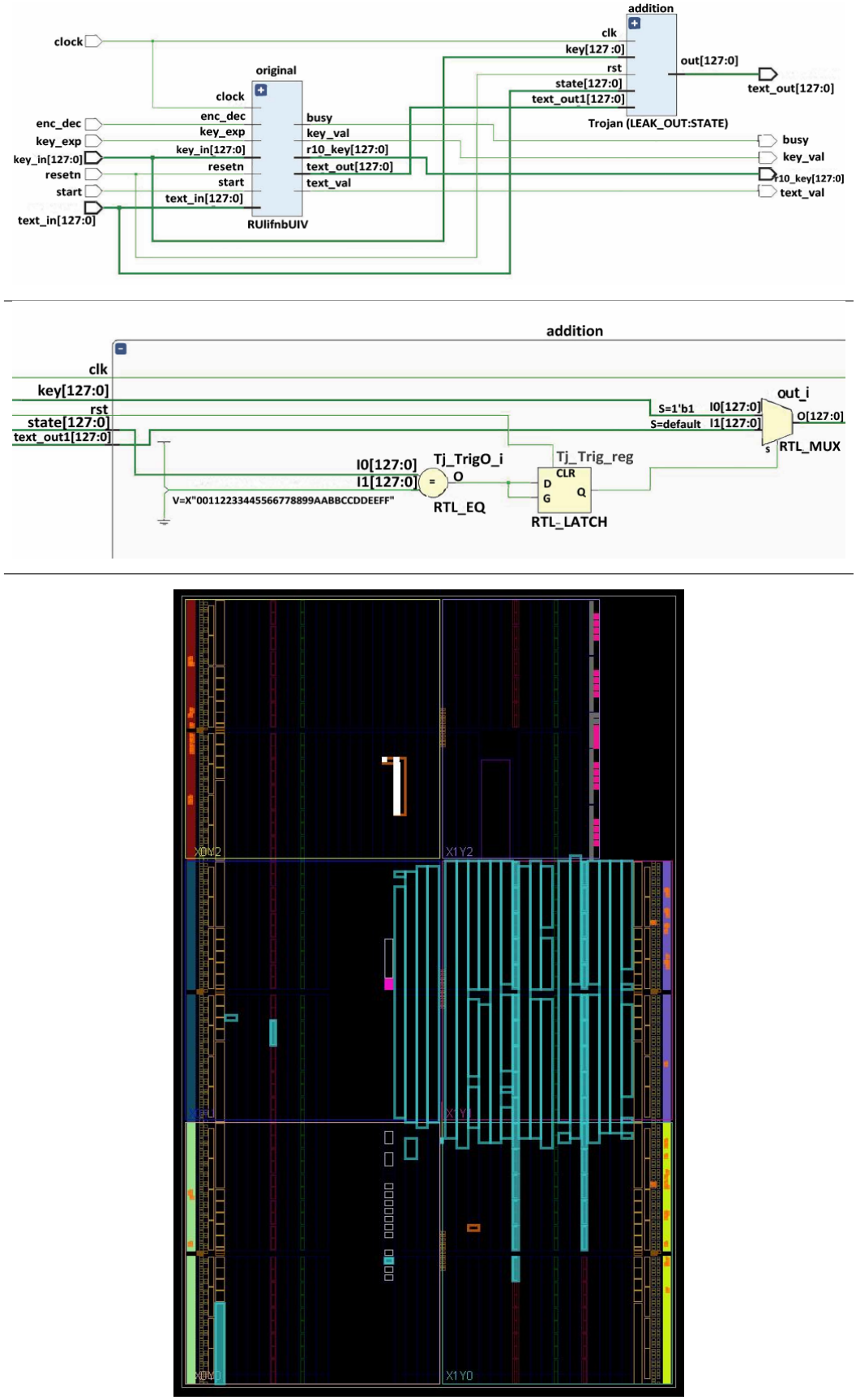

Figure 2. (**Top**): block diagram of the AES with the Trojan LEAK_OUT:STATE; (**middle**): the trigger mechanism of Trojan LEAK_OUT:STATE; and (**bottom**): the implemented layout (where the inserted Trojan is highlighted in white).

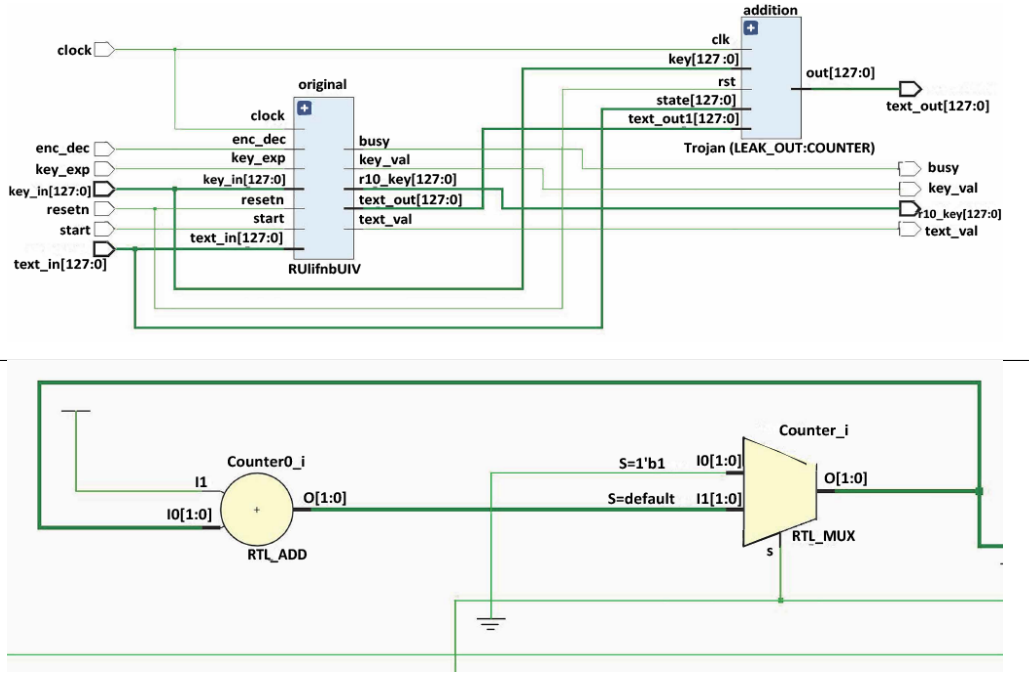

Figure 3. (**Top**): block diagram of the AES with Trojan LEAK_OUT:COUNTER; (**bottom**): the trigger mechanism of Trojan LEAK_OUT:COUNTER.

The insertion of Trojan LEAK_OUT:SEQ using the FEINT tool is shown in Figure 4 in the top section. Similar to LEAK_OUT:STATE and LEAK_OUT:COUNTER, the signals that are involved while inserting the Trojan are clock, key_in, resetn, text_in, and text_out. The trigger in this Trojan is a sequential trigger. This trigger waits for four separate values of plain text to be observed, in order, before setting Tj_Trig to 0x1. The inputs are as follows: 0x3243F6A8 885A308D 313198A2 E0370734, 0x00112233 44556677 8899AABB CCDDEEFF, 0x0, and 0x1. Note that these inputs do not need to be observed in immediate succession. Figure 4 in the bottom section shows the first two state comparisons. The register State0_reg identifies whether the first triggering input combination has been seen since the last reset. This Trojan has the same functionality as LEAK_OUT:STATE and LEAK_OUT:COUNTER. This Trojan overwrites the output text_out with the value of key_in when activated.

Figure 5 shows the block diagram of AES with Trojan LEAK_SIDE:STATE. In this example, the output signal is not rerouted. Here, the Trojan leaks the key through a side channel. Hence, an extra signal (capacitance) is introduced into the module. The signals of AES, clock, key_in, resetn, and text_in are joined with the corresponding signals of the LEAK_SIDE:STATE module. The trigger in this Trojan is identical to that of LEAK_OUT:STATE. The Trojan is activated when 0x00112233 44556677 8899AABB CCDDEEFF is observed on the input bus. The LFSR register is rotated every clock cycle, but only after activation, as represented by Tj_Trig = 1. The initial value of the LFSR is taken from the input plain text. AES-T1000 takes its initial value from the incoming plain text, and this value is loaded at reset.

A block diagram of the AES with Trojan DOS_OUT:SEQ is shown in Figure 6. The signal manipulation in this insertion is similar to that of LEAK_OUT:STATE. The signals clock, key_in, resetn, and text_in are joined with the corresponding signals of the Trojan module, while the signal text_out is rerouted through the Trojan module. The trigger in this Trojan is similar to that of LEAK_OUT:SEQ. The Trojan is activated after observing a sequence of patterns on the input bus. This Trojan is designed to perform a denial of service. When activated, the Trojan sends all zeros to the output instead of the encrypted output.

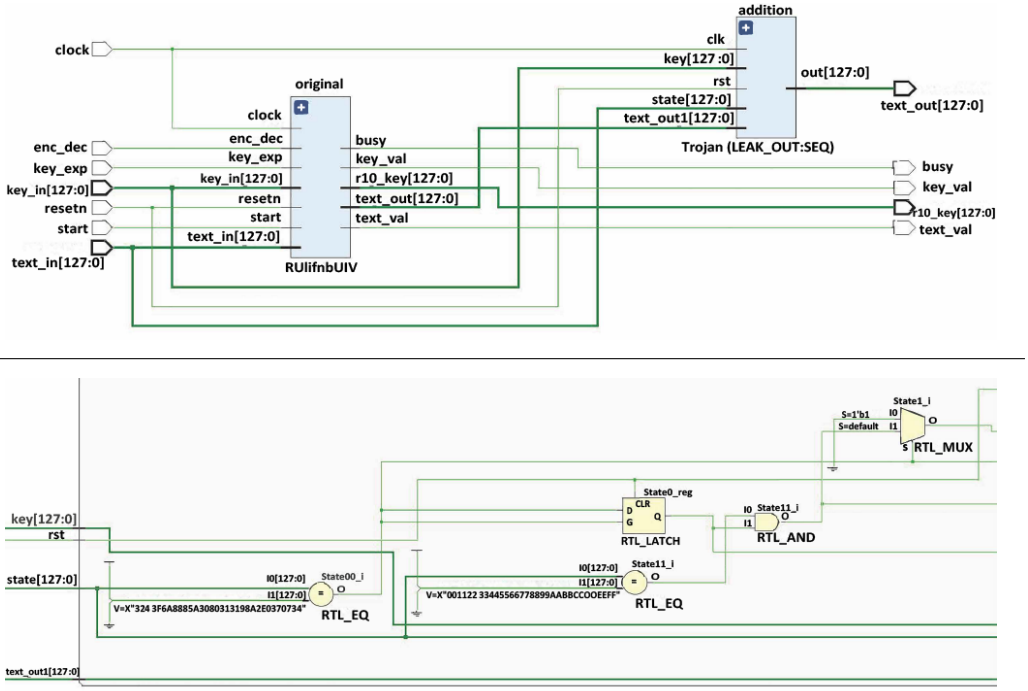

Figure 4. (**Top**): block diagram of the AES with Trojan LEAK_OUT:SEQ; (**bottom**): the trigger mechanism of Trojan LEAK_OUT:SEQ.

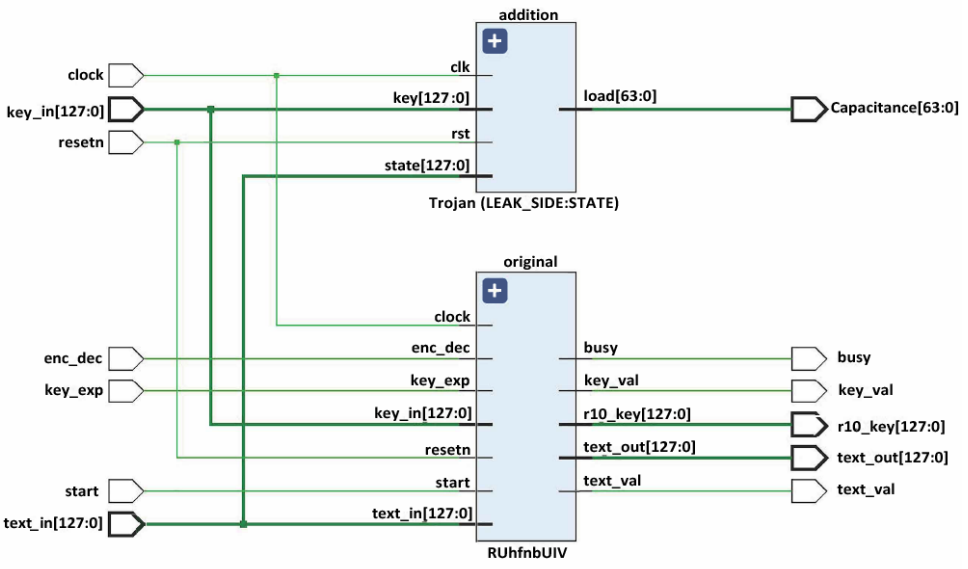

Figure 5. Block diagram of the AES with Trojan LEAK_SIDE:STATE.

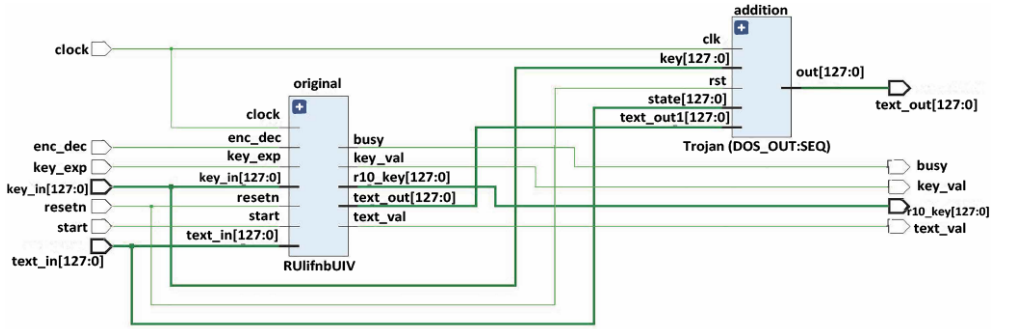

Figure 6. Block diagram of the AES with Trojan DOS_OUT:SEQ.

Validation: To validate the performance of FEINT, we considered two aspects:

- RTL/netlist file size increases after Trojans are inserted: Prior to Trojan insertion, the RTL file size represents the original design specifications and is generally determined by the complexity of the design's logic, number of interconnected components, and amount of control and data paths. After the insertion of a Trojan, the RTL files will likely see an increase in size due to the addition of new logic components required for the Trojan's functionality. The extent of the size increase depends on the complexity of the Trojan and its integration into the existing design. Table 3 presents information for each Trojan in Table 2 in terms of bytes for the RTL designs and netlist files. Each row of the table represents a different Trojan, and the associated sizes are provided for both RTL and netlist representations. The first row of the table is for the clean (no-Trojan) design. In the subsequent rows, the second and third column (i.e., "RTL Size" and "Netlist Size (Trojan inserted at RTL)" columns) are the RTL size and synthesized netlist size when the Trojan is inserted into the RTL. The fourth column (i.e., "Netlist Size (Trojan inserted at netlist)") is the combined netlist size when the Trojan is inserted directly into the netlist. It was noted that the observed file sizes were in line with what would be obtained if the required edits for inserting the Trojans were performed "by hand" instead of using FEINT, thus validating that FEINT does indeed efficiently perform the user-specified operations for editing signals and combining original and template/Trojan modules with an automated tool chain.
- Correctness of Trojan insertions: This was verified by checking that the outputs were unaffected when the trigger was not applied and that the Trojan behavior manifested when the trigger was applied. Depending on the particular Trojan used, the Trojan behavior could be, for example, AES secret key leakage or a DoS attack on the AES core. The validation process on our hardware testbed verified both the Trojan effect and the trigger for all the cases shown in Table 2 based on the specific details for each Trojan case, as described above. For example, for the LEAK_OUT:STATE Trojan, Figure 7 shows the Trojan effect (i.e., a leaking of the key as the output instead of the cipher text) when the Trojan trigger was applied at the third encryption. The plain text 0x00112233445566778899AABBCCDDEEFF was set as the trigger and, as mentioned earlier in this section, the secret key in our AES experiments was 0x12233445566778899AABBCCDDEEFF001.

```
 * Starting main loop*
Key Value:       0x12233445566778899AABBCCDDEEFF001

 * Running Encrypt 1 *
Plaintext:       0x8BADF00DC00010FFDEADC0DE00ABCDEF
Ciphertext:      0x6ED8AAD879E0C3A101732FB8B2364402

 * Running Encrypt 2 *
Plaintext:       0x9B18503614429A1F48E3E465E556A522
Ciphertext:      0x110A49DB01D5E837812FC6652E9CC931

 * Running Encrypt 3 (Trojan 1 Trigger) *
Plaintext:       0x00112233445566778899AABBCCDDEEFF
Ciphertext:      0x12233445566778899AABBCCDDEEFF001

 * Running Encrypt 4 *
Plaintext:       0xB35B56C535D906AE2E76C95051F91D2B
Ciphertext:      0xD6E2D445057CBE9E3DE842198DE86E70

 * Running Encrypt 5 *
Plaintext:       0x8BADF00DC00010FFDEADC0DE00ABCDEF
Ciphertext:      0x6ED8AAD879E0C3A101732FB8B2364402
```

Figure 7. Trojan LEAK_OUT:STATE insertion: key leakage observed during third encryption when the Trojan trigger was applied.

Table 3. File sizes of the AES crypto-accelerator before and after Trojan insertion.

Trojan	RTL Size	Netlist Size (Trojan Inserted at RTL)	Netlist Size (Trojan Inserted at Netlist)
No Trojan	40,869 bytes	1,171,519 bytes	1,171,519 bytes
LEAK_OUT:STATE	41,433 bytes	1,466,871 bytes	1,266,863 bytes
LEAK_OUT:COUNTER	41,688 bytes	1,457,615 bytes	1,296,571 bytes
LEAK_OUT:SEQ	41720 bytes	1,557,058 bytes	1,306,123 bytes
LEAK_SIDE:STATE	44,267 bytes	1,440,887 bytes	1,180,288 bytes
DOS_OUT:SEQ	41,711 bytes	1,552,688 bytes	1,288,838 bytes

Remark on Trojan detection methods: In the context of the above discussion on the possible application of FEINT to Trojan insertion, it is worthwhile to briefly summarize some potential methods to detect malicious modifications, even though such methods are orthogonal to the problem and scope addressed in this paper. One approach for detecting the presence of hardware Trojans is via functional testing. The primary challenge lies in defining an effective set of test vectors to uncover the unknown Trojans. Advanced testing techniques include methods such as [31], which use the inverted outputs of flip flops to expand the state space and improve detection probability. Randomization [32], and genetic algorithm-based methods [33] could also increase the likelihood of Trojan activation. Alternatively, side-channel analysis can be applied by leveraging the physical characteristics of the device, such as power consumption, electromagnetic radiation, and timing delays, to detect anomalies indicative of hardware Trojans. Approaches include the use of Principal Component Analysis (PCA) [34], leakage current analysis [35,36], short-term aging [9,37,38], and dynamic current monitoring [39,40].

4.2.2. Defender's Perspective

In order to showcase the utility of the FEINT tool from a defender's perspective, we employ it to insert a comprehensive BIST architecture into the existing AES core design. Figure 8 illustrates the integration of the BIST controller at the output of the AES core. This BIST framework incorporates seven of the fifteen tests specified in the National Institute of Standards and Technology (NIST) Statistical Test Suite for Random and Pseudo-Random Number Generators for Cryptographic Applications.

The inclusion of these tests empowers the defender to assess the integrity of the AES core by analyzing the randomness of the generated cipher texts. For each of these tests, a random plain text and secret key are defined, and the resulting cipher text is looped back into the system as the succeeding plain text for the subsequent encryption event. In this manner, the sequence of bits in the cipher text undergoes analysis by the BIST through various statistical tests, thus allowing for evaluations of randomness. Importantly, this approach leverages the well-established assessment of randomness in feedback-looped AES cores, thus enabling the verification of the implemented module's integrity. Any alterations that may impact the output of the core will subsequently affect the assessed randomness [41]. The seven specific tests employed in this proof of concept, along with their corresponding criteria for randomness evaluation, are detailed in Table 4.

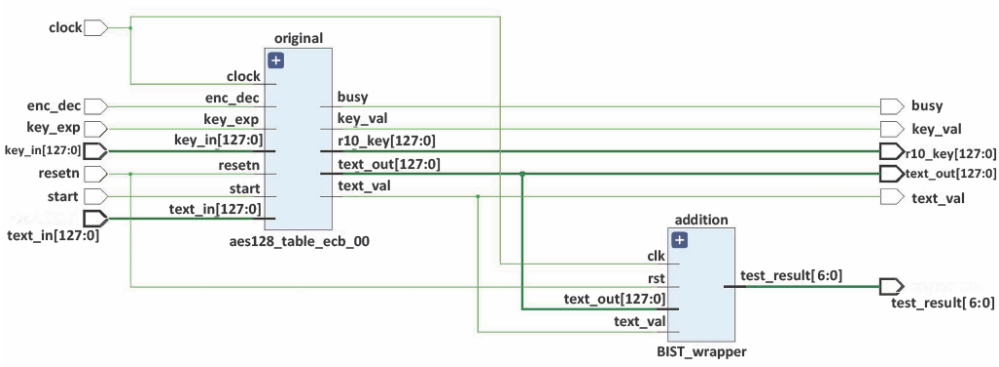

Figure 8. AES design alongside an inserted BIST template.

Table 4. BIST templates inserted using the FEINT tool.

Test Name	Evaluation Criteria
Frequency	Proportion of zeros and ones for entire test space
Frequency (Block)	Proportion of zeros and ones for 128-bit blocks
Runs	Number of sequences of identical bits
Longest Run of Ones	Length of longest run of ones vs. expected value
Non-Overlapping	Detection of occurrences of aperiodic pattern
Overlapping	Detection of occurrences of pre-specified string
Cumulative Sums	Maximum excursion of a random walk

4.2.3. Designer's Perspective

The versatility of the FEINT tool is demonstrated from the perspective of a hardware designer, showcasing its application in the seamless integration of commonly used, pre-defined circuit templates. As shown in this illustration, we employed the FEINT tool to insert pseudo-random number generator (PRNG) templates into an existing AES core design, leveraging these templates to generate secret keys. Figure 9 provides a visual representation of the AES circuit with the integrated PRNG templates. For this scenario, we utilized four distinct PRNG templates, as detailed in Table 5.

Table 5. PRNG templates inserted using the FEINT tool.

PRNG Name	Description
LFSR	RNG algorithm based on feedback shift register
Xoroshiro128+	RNG algorithm developed in 2016
Mersenne Twister	RNG algorithm developed in 1997
Trivium	RNG algorithm based on Trivium Stream Cipher

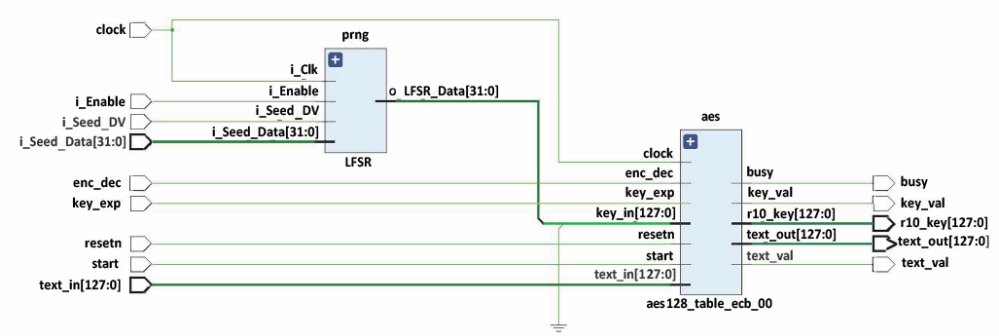

Figure 9. AES design alongside an inserted RNG template.

4.3. Back End Application

The placement of the inserted logic was considered for all three case studies. The constraints involving the placement of each of the inserted designs were mostly uniform, employing the same Vivado Xilinx Design Constraint (XDC) commands.

The variations in XDC commands stemmed from the Vivado-generated cell names, which were made uniform and predictable using the aforementioned synthesis attributes placed within the RTL. The chosen synthesis attributes resulted in cell and leaf names that closely followed the module name, and these were followed by a number. An example use of the `DONT_TOUCH` synthesis attribute, which was used in all case studies, is shown in Listing 3. Once the cell and leaf names were made predictable and uniform, the XDC file was appended with the desired insertion location. Listing 4 depicts the placement of a LUT to Location D6 within `SLICE_X13Y126` of the FPGA, which is then followed by a region constraint placing the `cpu` block to the locations specified by coordinates `SLICE_X28Y41:SLICE_X65Y99`. The implemented locations after the place-and-route phase of the design flow can be seen in Figure 2 in the bottom section, where the newly inserted design, highlighted in white, is constrained to the upper-left clock region, and the original design containing the `cpu` module, highlighted in blue, is constrained to the middle-right clock region.

Listing 3. Verilog Example of Trojan Module with Synthesis Attribute.

```
(* DONT_TOUCH = "true" *) module Trojan(
input           clk,
input           rst,
input   [127:0] state,
input   [127:0] key,
input   [127:0] text_out1,
output  [127:0] out
);

```

Listing 4. Vivado Example of Hybrid Location and Region XDC Constraint.

```
set_property LOC SLICE_X13Y126 [get_cells u_ctrl0/carry_i]
set_property BEL D6LUT [get_cells u_ctrl0/carry_i]

create_pblock PBlock_CPU
add_cells_to_pblock [get_pblocks PBlock_CPU] [get_cells -quiet [list cpu i2cp]]
resize_pblock [get_pblocks PBlock_CPU] -add {SLICE_X28Y41:SLICE_X65Y99}

```

5. Conclusions and Future Works

This work introduces a platform-independent tool called FEINT, which simplifies the process of module composition and automates necessary design-level modifications when incorporating new modules into an existing design. FEINT functions as a template insertion tool, using a user-provided configuration script to introduce dynamic design features as plugins at different stages of FPGA design, thus facilitating rapid prototyping, evolution through composition-based design, and system customization. The proposed tool is especially valuable in scenarios where designers want to tailor system behavior without requiring advanced FPGA programming skills, and it enables reducing manual effort for customizing template/Trojan instantiation and insertion. Moreover, FEINT is scalable, future-proof, compatible with different FPGA families and tool versions, and it can be seamlessly integrated with commercial tools. We have demonstrated the efficacy of our tool by depicting its use in different scenarios when inserting templates into an already existing AES cryptographic core. These example scenarios, based on different insertion objectives, included the use of the framework as a means to insert Trojans as an attacker, a BIST as a defender, and a RNG/LFSR as a designer. As a future direction, we aim to enhance our tool by integrating machine learning for greater efficiency. It is also interesting to explore the potential for developing a similar EDA flow for Application-Specific Integrated Circuits (ASICs) as a research avenue.

Author Contributions: Conceptualization, P.K., J.T., R.K. and F.K.; methodology, V.R.S., H.P., P.K., R.K. and F.K.; software, V.R.S., R.S. and M.R.; validation, V.R.S., R.S., M.R., H.P. and P.K.; formal analysis, V.R.S., H.P. and P.K.; investigation, V.R.S., R.S., M.R., H.P. and P.K.; resources, P.K., R.K. and F.K.; data curation, V.R.S., R.S., M.R., H.P. and P.K.; writing—original draft preparation, V.R.S., R.S., M.R., H.P. and P.K.; writing—review and editing, V.R.S., R.S., M.R., H.P., P.K., J.T., R.K. and F.K.; visualization, V.R.S., R.S. and M.R.; supervision, P.K., J.T., R.K. and F.K.; project administration, P.K., J.T., R.K. and F.K.; funding acquisition, P.K., J.T., R.K. and F.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported, in part, by DoE Kansas City. Honeywell Federal Manufacturing & Technologies, LLC operates the Kansas City National Security Campus for the United States Department of Energy/National Nuclear Security Administration under contract number DE-NA0002839.

Data Availability Statement: Data available on request from the authors.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kuon, I.; Rose, J. *Quantifying and Exploring the Gap between FPGAs and ASICs*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2010.
2. Trimberger, S.M. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proc. IEEE* **2015**, *103*, 318–331. [CrossRef]
3. Caulfield, A.M.; Chung, E.S.; Putnam, A.; Angepat, H.; Fowers, J.; Haselman, M.; Heil, S.; Humphrey, M.; Kaur, P.; Kim, J.Y.; et al. A cloud-scale acceleration architecture. In Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–13. [CrossRef]
4. Cheng, Q.; Huang, M.; Man, C.; Shen, A.; Dai, L.; Yu, H.; Hashimoto, M. Reliability Exploration of System-on-Chip with Multi-Bit-Width Accelerator for Multi-Precision Deep Neural Networks. *IEEE Trans. Circuits Syst. Regul. Pap.* **2023**, *70*, 3978–3991. [CrossRef]
5. Zhang, C.; Prasanna, V. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 35–44.
6. Waksman, A.; Sethumadhavan, S. Silencing hardware backdoors. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, 22–25 May 2011; IEEE: Piscataway, NJ, USA, 2011; pp. 49–63.
7. Lin, L.; Kasper, M.; Güneysu, T.; Paar, C.; Burleson, W. Trojan Side-Channels: Lightweight Hardware Trojans through Side-Channel Engineering. In Proceedings of the Cryptographic Hardware and Embedded Systems, CHES '09, Berlin/Heidelberg, Germany, 6–9 September 2009; pp. 382–395.
8. Elnaggar, R.; Chaudhuri, J.; Karri, R.; Chakrabarty, K. Learning Malicious Circuits in FPGA Bitstreams. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2023**, *42*, 726–739. [CrossRef]
9. Surabhi, V.R.; Krishnamurthy, P.; Amrouch, H.; Henkel, J.; Karri, R.; Khorrami, F. Exposing Hardware Trojans in Embedded Platforms via Short-Term Aging. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3519–3530. [CrossRef]
10. Zhao, W.; Shen, H.; Li, H.; Li, X. Hardware Trojan Detection Based on Signal Correlation. In Proceedings of the IEEE Asian Test Symposium, Hefei, China, 15–18 October 2018; pp. 80–85. [CrossRef]
11. Standaert, F.X.; Örs, S.B.; Preneel, B. Power Analysis of an FPGA. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2004, Cambridge, MA, USA, 11–13 August 2004; Joye, M., Quisquater, J.J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 30–44.
12. Moradi, A.; Barenghi, A.; Kasper, T.; Paar, C. On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks: Extracting Keys from Xilinx Virtex-II FPGAs. In Proceedings of the ACM Conference on Computer and Communications Security, New York, NY, USA, 17–21 October 2011; CCS '11, pp. 111–124. [CrossRef]
13. Krautter, J.; Gnad, D.R.E.; Tahoori, M.B. FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. *IACR Trans. CHES* **2018**, *2018*, 44–68. [CrossRef]
14. Glamočanin, O.; Mahmoud, D.G.; Regazzoni, F.; Stojilović, M. Shared FPGAs and the Holy Grail: Protections against Side-Channel and Fault Attacks. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; pp. 1645–1650. [CrossRef]
15. Bilgin, B.; Gierlichs, B.; Nikova, S.; Nikov, V.; Rijmen, V. Higher-Order Threshold Implementations. In Proceedings of the Advances in Cryptology—ASIACRYPT 2014, Kaoshiung, Taiwan, 7–11 December 2014; Sarkar, P., Iwata, T., Eds.; pp. 326–343.
16. Gross, H.; Mangard, S.; Korak, T. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. Cryptology ePrint Archive, Paper 2016/486. 2016. Available online: https://eprint.iacr.org/2016/486 (accessed on 17 September 2023).
17. Lohrke, H.; Tajik, S.; Krachenfels, T.; Boit, C.; Seifert, J.P. Key Extraction Using Thermal Laser Stimulation: A Case Study on Xilinx Ultrascale FPGAs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2018**, *2018*, 573–595. [CrossRef]

18. Yoon, J.; Seo, Y.; Jang, J.; Cho, M.; Kim, J.; Kim, H.; Kwon, T. A Bitstream Reverse Engineering Tool for FPGA Hardware Trojan Detection. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Toronto, ON, Canada, 15–19 October 2018; pp. 2318–2320. [CrossRef]
19. Mardani Kamali, H.; Zamiri Azar, K.; Gaj, K.; Homayoun, H.; Sasan, A. LUT-Lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Hong Kong, China, 8–11 July 2018; pp. 405–410. [CrossRef]
20. Olney, B.; Karam, R. Tunable FPGA Bitstream Obfuscation with Boolean Satisfiability Attack Countermeasure. *ACM Trans. Des. Autom. Electron. Syst.* **2020**, *25*, 1–22. [CrossRef]
21. Subramanyan, P.; Ray, S.; Malik, S. Evaluating the security of logic encryption algorithms. In Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 5–7 May 2015; pp. 137–143. [CrossRef]
22. Yasin, M.; Mazumdar, B.; Rajendran, J.J.V.; Sinanoglu, O. SARLock: SAT attack resistant logic locking. In Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST), McLean, VA, USA, 3–5 May 2016; pp. 236–241. [CrossRef]
23. Yasin, M.; Rajendran, J.J.; Sinanoglu, O.; Karri, R. On Improving the Security of Logic Locking. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2016**, *35*, 1411–1424. [CrossRef]
24. Lavin, C.; Kaviani, A. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In Proceedings of the IEEE Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 29 April–1 May 2018; pp. 133–140. [CrossRef]
25. Backasch, R.; Hempel, G.; Werner, S.; Groppe, S.; Pionteck, T. Identifying homogenous reconfigurable regions in heterogeneous FPGAs for module relocation. In Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, Mexico, 8–10 December 2014; pp. 1–6. [CrossRef]
26. Athanas, P.; Bowen, J.; Dunham, T.; Patterson, C.; Rice, J.; Shelburne, M.; Suris, J.; Bucciero, M.; Graf, J. Wires on Demand: Run-Time Communication Synthesis for Reconfigurable Computing. In Proceedings of the International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27–29 August 2007; pp. 513–516. [CrossRef]
27. Cruz, J.; Posada, C.; Masna, N.V.R.; Chakraborty, P.; Gaikwad, P.; Bhunia, S. A Framework for Automated Exploration of Trojan Attack Space in FPGA Netlists. *IEEE Trans. Comput.* **2023**, *72*, 2740–2751. [CrossRef]
28. Fyrbiak, M.; Wallat, S.; Swierczynski, P.; Hoffmann, M.; Hoppach, S.; Wilhelm, M.; Weidlich, T.; Tessier, R.; Paar, C. HAL—The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IEEE Trans. Dependable Secur. Comput.* **2019**, *16*, 498–510. [CrossRef]
29. Jyothi, V.; Krishnamurthy, P.; Khorrami, F.; Karri, R. TAINT: Tool for Automated INsertion of Trojans. In Proceedings of the IEEE International Conference on Computer Design (ICCD), Boston, MA, USA, 5–8 November 2017; pp. 545–548.
30. Intel. Intel FPGA Design Flow. Available online: https://www.intel.com/content/www/us/en/docs/programmable/683562/21-3/introduction-to-fpga-design-flow-for-users.html (accessed on 17 September 2023).
31. Banga, M.; Hsiao, M.S. ODETTE: A non-scan design-for-test methodology for Trojan detection in ICs. In Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust, San Diego, CA, USA, 5–6 June 2011; pp. 18–23. [CrossRef]
32. Jha, S.; Jha, S.K. Randomization Based Probabilistic Approach to Detect Trojan Circuits. In Proceedings of the IEEE High Assurance Systems Engineering Symposium, Nanjing, China, 3–5 December 2008; pp. 117–124. [CrossRef]
33. Shi, Z.; Ma, H.; Zhang, Q.; Liu, Y.; Zhao, Y.; He, J. Test generation for hardware trojan detection using correlation analysis and genetic algorithm. *ACM Trans. Embed. Comput. Syst. TECS* **2021**, *20*, 1–20. [CrossRef]
34. Agrawal, D.; Baktir, S.; Karakoyunlu, D.; Rohatgi, P.; Sunar, B. Trojan Detection using IC Fingerprinting. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 20–23 May 2007; pp. 296–310. [CrossRef]
35. Aarestad, J.; Acharyya, D.; Rad, R.; Plusquellic, J. Detecting Trojans Through Leakage Current Analysis Using Multiple Supply Pad I_{DDQ}s. *IEEE Trans. Inf. Forensics Secur.* **2010**, *5*, 893–904. [CrossRef]
36. Rad, R.; Plusquellic, J.; Tehranipoor, M. Sensitivity analysis to hardware Trojans using power supply transient signals. In Proceedings of the IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim, CA, USA, 9 June 2008; pp. 3–7. [CrossRef]
37. Surabhi, V.R.; Krishnamurthy, P.; Amrouch, H.; Basu, K.; Henkel, J.; Karri, R.; Khorrami, F. Hardware Trojan Detection Using Controlled Circuit Aging. *IEEE Access* **2020**, *8*, 77415–77434. [CrossRef]
38. Surabhi, V.R.; Krishnamurthy, P.; Amrouch, H.; Henkel, J.; Karri, R.; Khorrami, F. Trojan Detection in Embedded Systems with FinFET Technology. *IEEE Trans. Comput.* **2022**, *71*, 3061–3071. [CrossRef]
39. Piliposyan, G.; Khursheed, S.; Rossi, D. Hardware Trojan Detection on a PCB Through Differential Power Monitoring. *IEEE Trans. Emerg. Top. Comput.* **2022**, *10*, 740–751. [CrossRef]
40. Salmani, H.; Tehranipoor, M. Layout-Aware Switching Activity Localization to Enhance Hardware Trojan Detection. *IEEE Trans. Inf. Forensics Secur.* **2012**, *7*, 76–87. [CrossRef]
41. Doulcier, M.; Flottes, M.L.; Rouzeyre, B. AES vs LFSR Based Test Pattern Generation: A Comparative Study. In Proceedings of the LATW: Latin American Test Workshop, Cuzco, Peru, 11–14 March 2007; pp. 314–321.