Assuring Netlist-to-Bitstream Equivalence using Physical Netlist Generation and Structural Comparison

Reilly McKendrick, Keenan Faulkner and Jeffrey Goeders
Department of Electrical and Computer Engineering
Brigham Young University
Provo, Utah, USA
{romckend, keenanrf, jgoeders}@byu.edu

2023 International Conference on Field Programmable Technology (ICFPT) | 979-8-3503-5911-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICFPT59805.2023.00021

***Abstract*—Hardware netlists are generally converted into a bitstream and loaded onto an FPGA board through vendor-provided tools. Due to the proprietary nature of these tools, it is up to the designer to trust the validity of the design's conversion to bitstream. However, motivated attackers may alter the CAD tools' integrity or manipulate the stored bitstream with the intent to disrupt the functionality of the design.**

This paper proposes a new method to prove functional equivalence between a synthesized netlist, and the produced FPGA bitstream. The novel approach is comprised of two phases: first, we show how we can utilize implementation information to perform a series of transformations on the netlist, which do not affect its functionality, but ensure it structurally matches what is physically implemented on the FPGA. Second, we present a structural mapping and equivalence checking algorithm that verifies this physical netlist exactly matches the bitstream. We validate this process on several benchmark designs, including checking for false positives by injecting hundreds of design modifications.

I. Introduction

When fabricating integrated circuits, such as ASICs and FPGAs, the designer must trust that the chip is fabricated correctly, and that the design is not modified by the fabricator. This is especially important for military and defense applications, or other security-sensitive applications; as such, these organizations will typically employ trusted foundries to ensure correct fabrication of their IC [1]. However, for FPGAs, ensuring correct "fabrication" goes beyond just the physical manufacturing of the chip; since FPGAs can be configured to implement arbitrary digital circuits, a security conscious designer must also ensure that the circuit design configured onto the FPGA is free from malicious hardware trojans.

Even if a designer ensures their RTL design is Trojan-free, ensuring the integrity of an FPGA design means that the designer must also trust the integrity of the CAD tools used to convert the designer-provided circuit description into the bitstream configuration file (i.e., the "digital fab"), and they must also trust that the bitstream is not modified post-compilation. While many FPGA users take this for granted, security conscious organizations may be concerned with the possible vulnerabilities that exist with this process. For example, a malicious actor with a CAD tool company may be able to modify the tool to silently inject back doors or kill switches into the design, an unknown bug in the tools may produce a design where certain internal signals are accidentally leaked, malware on a designer's computer may replace portions of an otherwise safe CAD flow with malicious tools that inject hardware trojans into the produced bitstream, or finally, the generated bitstream could be intercepted and modified post-compilation.

Given these concerns, designers may wish to verify that a given FPGA bitstream does in fact exactly match the circuit description that they provided to the CAD tools. Unfortunately, providing this assurance is very challenging. The original circuit description goes through many optimizations and transformations as it is mapped and implemented on the FPGA, and furthermore, the bitstream itself is proprietary and not in any way human readable. No commercial or academic tools presently exist that can easily verify equivalence between an RTL design and the produced bitstream.

In this work we present a novel approach for assuring equivalence between a user's design and the FPGA bitstream. Our technique is illustrated in Figure 1. While an ideal equality check would compare the designer's RTL to the produced bitstream, comparing against the original RTL is very challenging due to transformations performed by the synthesis tool [2]. Instead, we present a technique to verify functional equivalence between the post-synthesis netlist and the FPGA bitstream. Our approach consists of two main phases: physical netlist generation and structural comparison. The physical netlist pass transforms the netlist to mimic transformations made to the design during the implementation stage of the CAD compilation. These transformations do not change the functional nature of the design, but they do ensure that the netlist exactly matches the physical implementation in the bitstream.

Next, a structural comparison pass establishes a mapping between the physical netlist and a netlist obtained from the

This work was funded by the Office of Naval Research, award N00014-22-1-2683.

979-8-3503-5911-4/23/$31.00 ©2023 IEEE
DOI 10.1109/ICFPT59805.2023.00021

bitstream. This *reversed netlist* will contain no hierarchy and no instance or signal names, making it challenging to compare against using traditional techniques such as logical equivalence checking offered by commercial tools [2]. However, since the structure of the reversed netlist will perfectly match the structure of the physical netlist, we can use an algorithm that leverages this information to make the comparison.

The major contributions of this work are:

- A physical netlist generator, that leverages RapidWright [3] to transform a logical netlist to a form that exactly matches the physical implementation in the bitstream. While in this work we propose using this netlist for assurance purposes, it may also be useful for other custom CAD tool and design analysis purposes.
- A structural comparison tool that demonstrates how cells and nets can be mapped between a physical netlist and a reverse-engineered netlist. This tool can be used to verify netlist-to-bitstream equivalence with better scalability than commercial functional equivalence checking tools.
- An open-source release of both of these tools, as part of the BYU FPGA Assurance Tool (BFASST), available at https://github.com/byuccl/bfasst.
- An experimental validation of the proposed flow on many benchmark designs, including checking for false positives by injecting design modifications into the reversed netlist and verifying that the equivalence check detects the changes.

II. Background

A. Trojan Detection

As custom hardware circuits continue to see use in defense and security-sensitive applications, there is a growing concern about the presence of malicious hardware trojans in these circuits [4]. Hardware trojans are malicious third-party modifications to circuits, and can take the form of back doors into circuits, kill switches, intentional leaks of sensitive information, or other harmful modifications. While evidence of hardware trojans in the wild is limited [4], [5], their potential for severe destructive capabilities has motivated much research into techniques for detecting them. There have been several works focused on detecting hardware trojans in digital circuits, including static analysis of the source design [6], [7], using embedded circuitry to monitor the system for anomalous behavior [8]–[12], or a combination of these techniques [13].

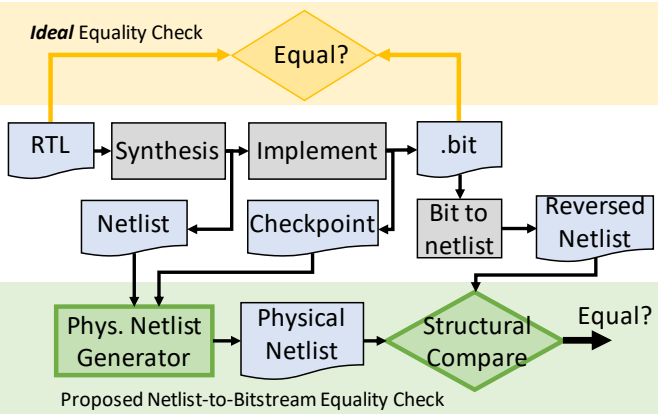

Fig. 1: Overview of the proposed assurance flow, which verifies equivalency between the post-synthesis netlist and the bitstream.

The proposed techniques in this paper are not focused on detecting hardware trojans, but rather on providing assurance that the bitstream exactly matches the user's design. Our proposed techniques would be used in conjunction with trojan detection techniques: a designer would first employ static analysis techniques to ensure their netlist is trojan-free, and then after generating a bitstream, use our proposed technique to ensure the produced bitstream remains equivalent to their input netlist.

B. Bitstream-Level Attacks

The proposed work is aimed at detecting cases where the produced FPGA bitstream is compromised, but the original circuit design remains safe and unchanged. As discussed previously, this could occur with compromised CAD tools, or it could occur if the FPGA bitstream was compromised post-generation. While these cases may seem improbable, recent work has shown both their possibility, and potential for significant consequences. In [14] and [15] the authors show how behavior of normal FPGA CAD tools can be exploited to insert trojan activation switches into the design during compilation. These changes are not detectable in the hardware source code, and require inspection of the FPGA bitstream, as proposed in this work. In [16] the authors show that it is possible to locate which portions of an FPGA bitstream control certain elements of an AES encryption module, thus allowing them to make modifications to the bitstream so that the final circuit is much easier to attack, and the encryption key can be obtained.

C. Related Work

There has so far been relatively little work in the research community on addressing equivalence of FPGA designs through the CAD flow.

In [17], Hastings et al. propose two different techniques to ensure an IP remains unmodified through the implementation phase of FPGA CAD. They first explore a physical-level approach that requires the IP to be pre-implemented in a design partition before being instanced in the user's design; however, this places the burden of implementation on the IP provider.

The other approach is more similar to our work in that it checks for functional equivalence from the netlist to the implemented design. However, their work does not perform any comparison against the bitstream, but rather just inspects the implementation details obtained from Vivado, and uses Cadence Conformal to compare to the post-synthesis netlist. This is a much easier problem to solve as signal names are maintained and can be used to seed the comparison in Conformal. However, this approach trusts that the CAD tool

self-reports correct design information, which goes against the premise of developing an assurance flow that verifies correctness of an untrusted CAD flow. Unlike this work, it also would fail to detect any attacks that occur during or after bitstream generations, such as those discussed in the previous subsection.

In [2], Cahill et al. expand upon this work and extend it to bitstream-level comparison. They leverage Project Icestorm [18], which provides bitstream-to-netlist tools for the Lattice iCE40 FPGAs (a 4-LUT architecture, sometimes referred to as the "world's smallest FPGA"). The comparison process is relatively successful, but encounters substantial scalability issues, as the commercial equivalence checkers they use are not designed to handle comparison between netlists with no seed mapping information (due to the absence of signal names in the reverse engineered netlist). The largest designs they test are a couple thousand 4-LUTs, with runtimes sometimes taking over 24 hours.

In contrast, this work presents a technique that is designed with scalability in mind. We do not rely on functional equivalence checkers where the state space can grow exponentially and become intractable. Rather, our approach of transforming the netlist to structurally match the bitstream means that we can perform a structural comparison between the two designs. Both netlists will have the exact same primitives connected in the exact same pattern, using the exact same set of pins. This makes for a much simpler comparison problem. In contrast, the equivalence checker tools used in [2], are trying to prove the much harder problem that despite the two netlists having different structures, they are still functionally equivalent.

III. Assurance Flow

Our netlist-to-bitstream assurance flow is shown in Figure 1. While we believe this approach could be applied to multiple FPGA vendors and device families, in this paper we present a proof-of-concept tool that targets the Xilinx 7-series FPGAs.

Unlike previous work [2], [17] that relied on custom synthesis tools or pre-implementation of the user design, our proposed flow allows the user to use standard CAD tools for synthesis and implementation. Xilinx Vivado is used to first synthesize the design to a post-synthesis (technology-mapped) netlist. Our current tool only works on flattened netlists, so the flattening option in Vivado must be set; however, with some additional engineering, our tool could be extended to support hierarchical netlists.

Since our assurance flow only compares the bitstream against the post-synthesis netlist, the user would need to ensure they are satisfied with the safety of this netlist. This would be accomplished through either trusting the commercial synthesis tool, using an in-house or open-source trusted synthesis tool, or vetting the post-synthesis netlist through trojan analysis techniques.

Next, the user can use Vivado to perform implementation and bitstream generation. During this process an implementation checkpoint needs to be exported out of Vivado. The produced bitstream is then converted to a netlist using Project X-Ray [19] to convert bitstream to FASM, and then fasm2bels [20] to convert FASM to a Verilog netlist. The netlist produced by fasm2bels is a completely flat netlist containing each BEL (Basic Element of Logic, such as lookup tables and registers) used in the bitstream, the BEL configuration properties, and the nets that connect the various BEL pins. The netlist contains no original internal signal or instance names; however, the original XDC constraint file provided to Vivado can also be provided to fasm2bels, which is sufficient to determine primary input and output signal names based on the pin constraints.

Once both a netlist and reversed engineered netlist are produced, our proposed assurance flow can proceed. It is comprised of two major steps: the physical netlist generation, and the structural mapping and comparison.

The physical netlist generation, detailed in Section IV, is a Python script that takes the post-synthesis netlist and the implementation checkpoint, and generates a transformed netlist that is structurally equivalent to the bitstream. The transformations are based on the implementation information provided by the Vivado implementation checkpoint, which is accessed using the RapidWright [3] tool. For convenience, RapidWright is also used to transform the netlist and generate the new netlist file, although it would be possible to do this using a fully open-source tool, such as Spydrnet [21].

The transformed netlist is then used as the golden netlist for comparison against the bitstream netlist. This comparison tool, described in Section V, is another Python script, and provides a structural comparison of the two netlists. It works by first establishing a mapping of each cell instance and net in the design, and then uses this mapping to verify the two designs are exactly identical.

IV. Physical Netlist Generator

This section describes our physical netlist generator, which takes a post-synthesis netlist and an implementation checkpoint, and produces a transformed netlist that is structurally equivalent to the bitstream.

One may suppose that if you export a netlist from Vivado *after* implementation, that it would already more closely resemble how the design is implemented in the bitstream; however, this is not the case. Even after implementation, when you export a netlist from Vivado you still are given the post-synthesis netlist.

A. Transformations

Our Python script utilizes RapidWright [3], which allows us to load the netlist and iterate through the design implementation data looking for places where implementation transformations have taken place. The following subsections discuss the implemented transformations that must be handled.

1) Replacing Logical Cells with Physical BEL Primitives: The post-synthesis netlist contains a collection of Xilinx *UNISIM Cells*; however, the UNISIM library of cells does not have a one-to-one relationship with physical BELs on the FPGA. For example, the UNISIM library contains *LUT1*,

```
BUFG clk_IBUF_BUFG_inst (
   .I(clk_IBUF),
   .O(clk_IBUF_BUFG));
```

(a) BUFG instance in original netlist

```
BUFGCTRL #(
   .INIT_OUT(0),
   .IS_CE0_INVERTED(1'b0),
   .IS_CE1_INVERTED(1'b1),
   .IS_IGNORE0_INVERTED(1'b1),
   .IS_IGNORE1_INVERTED(1'b0),
   .IS_S0_INVERTED(1'b0),
   .IS_S1_INVERTED(1'b1),
   .PRESELECT_I0("TRUE"),
   .PRESELECT_I1("FALSE"))
   clk_IBUF_BUFG_inst_phys (
      .CE0(1'b1), .CE1(1'b1),
      .I0(clk_IBUF), .I1(1'b1),
      .IGNORE0(1'b1), .IGNORE1(1'b1),
      .O(clk_IBUF_BUFG),
      .S0(1'b1), .S1(1'b1));
```

(b) BUFGCTRL instance generated in physical netlist

Fig. 2: BUFG cell replacement with BUFGCTRL BEL instance

LUT2, *LUT3*, *LUT4*, *LUT5*, and *LUT6* cells, depending on the size of the logic function, but physically, the 7-series FPGAs only have *LUT6_2* BELs, and all of these cells are mapped to this single type of BEL. Thus, for every *LUT** cell in the netlist, we remove it, and replace it with a *LUT6_2* instance.

Swapping out a smaller LUT for a larger one requires you to scale up the INIT property to a larger width, duplicating the INIT values for each additional LUT input. For example, a *LUT2* implementing a simple AND-gate would have a 4-bit INIT value of `4'h8`, but if replaced by a *LUT3*, the new INIT value would need to be `8'h88` (all LUTs are actually replaced with LUT6_2 instances, but we include this smaller LUT size replacement as a simpler example).

The need to replace UNISIM cells with cell instances that match physical BELs does not just occur for LUTs. For example, a *BUFG*, *BUFGCTRL*, and other *BUFG** cells are all mapped to the same *BUFGCTRL* BEL, and need to be replaced in the netlist. The *BUFGCTRL* has additional parameters and inputs not found on a *BUFG* cell, which need to be driven with appropriate constants to match the correct cell behavior. Figure 2 provides an example of this transformation taken from one of our test designs.

2) Logical to Physical Pin Mapping: In many cases FPGA primitives have *equivalent* pins, meaning that it doesn't matter which input a signal is connected to, as long as it is connected to one of the equivalent pins. For example, with a *LUT* primitive, all input pins are equivalent, since it doesn't matter the order in which the input signals are connected to the

Logical LUT Function

I0	I1	OUT
0	0	0
0	1	0
1	0	1
1	1	0

Physical LUT Function

I0=I1	I1=I0	OUT
0	0	0
0	1	1
1	0	0
1	1	0

Fig. 3: LUT pin reordering example

LUT pins. If the ordering of input pins are changed, the configured logic function can be modified appropriately. This behavior makes it easier for the router to route signals to LUT inputs, since the signal does not need to reach a specific LUT input, but rather any LUT input. This leads to the CAD tools frequently changing the ordering of the LUT inputs from the logical ordering in the netlist, to the physical ordering implemented on the FPGA.

To accommodate for this behavior, after replacing all LUTs with LUT6_2 primitives, we then reorder the inputs to match the logical to physical pin mapping, which is available through convenient functions in RapidWright. The INIT property must also be reordered to match the new pin ordering. Figure 3 shows an example of this transformation. The left-hand side of the figure shows the logic function $OUT = I0\&\sim I1$, while the right-hand side shows the logic function when the physical implementation has swapped the input pins, which now becomes $OUT = \sim I0\&I1$. The bits in the INIT value need to be shuffled accordingly, and in this simple example would be updated from `4'h4` to `4'h2`.

To make this change we leverage the RapidWright *LUTTools* class which contains functions to convert between equation strings and INIT values. We first convert the INIT value to an equation, then do a find and replace on the input names in the equation, and then convert back to an INIT value.

In our testing we have only observed Vivado changing pin orderings for LUT primitives. It is conceivable that other primitives may also undergo pin swapping (eg. swapping the two sets of input pins to a CARRY4 primitive); however, we have yet to observe this behavior.

3) LUT Combining: Another very common transformation that takes place is the combining of two logical LUTs into a physical *LUT6_2* primitive. Figure 4 provides a diagram of the *LUT6_2* primitive. As shown, a *LUT6_2* actually has two outputs, and can implement two logic functions from the netlist, provided that between the two logic functions, they are five or fewer unique input signals.

In our physical netlist transformation code we look for instances in the implemented design where this occurs, and extract which two LUTs from the netlist have been combined together. We then remove these two LUTs from the netlist and replace them with a single LUT6_2 instance. Care must

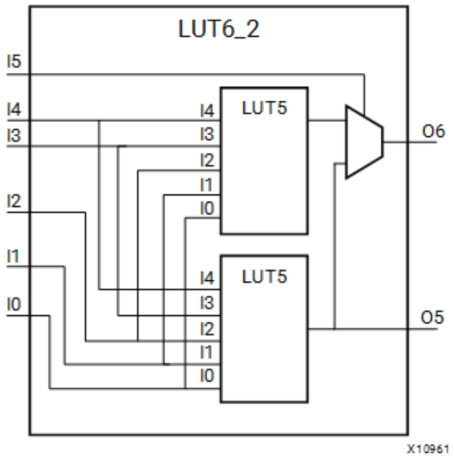

Fig. 4: Two LUT5s within a larger LUT6_2 primitive [22].

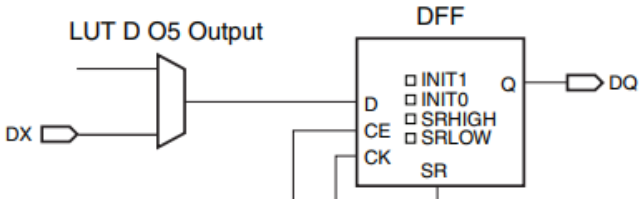

Fig. 5: Routing to a flip-flop through a LUT6_2 [23]

.

be taken to handle the mapping of logical to physical pins, as described in the previous section, and both sets of logical input pins must be combined and connected to the single new LUT6_2 instance. The INIT property must also be combined correctly, incorporating the technique previously described to produce two different 32-bit LUT5 INIT values that are then concatenated together to form the 64-bit LUT6_2 INIT value.

4) LUT Routethrus: In the 7-series FPGA architecture, each flip-flop is driven by a mux that selects between an output of the associated LUT6_2, and a dedicated CLB input for that flip-flop (eg. *DX* for the *DFF* BEL).

Typically if the flip-flop is not driven by a LUT placed at the associated LUT6_2 BEL, then the input signal will be routed to use the dedicated CLB input pin (eg. *DX*). However, on occasion the CAD tools will elect to route the signal to the flip-flop by going through the associated LUT6_2. This likely occurs because the router had an easier time routing to a LUT input (it could use *any* LUT input), than the dedicated flip-flop input pin. In this case the LUT is now playing an active role in the design, where there would have originally been no LUT to represent this buffered connection in the original netlist.

To handle this we add a new LUT6_2 instance to the design, disconnect the net that was driving the flip-flop and instead connect it to the appropriate LUT input, and then create a new net to connect from the LUT output to the flip-flop input. The LUT INIT must also be configured correctly to pass the appropriate input pin to the output.

This is further complicated by the fact that this new LUT routethru may share half of a LUT6_2 with another logic function in the netlist, for the reason described in the previous section. This edge case must be detected and handled appropriately.

An observant reader may now realize that how we have designed our physical netlist generator is quite dependent on the behavior of the bitstream-to-netlist tool we are using (fasm2bels). We need to add the LUT routethru as a new LUT instance in our design because the output of fasm2bels also outputs a LUT instance when it encounters a LUT routethru, and we want the designs to be structurally identical. The fasm2bels tool produces a netlist that closely represents the bitstream from which it was derived, and as such, doesn't make any attempt to optimize out LUTs that are only functioning as buffers. However, should a different tool be used to generate the reverse-engineered netlist, different design choices may need to be employed in the physical netlist generator.

5) LUT Constant Generators: Another implementation optimization we encountered was LUT logic being used to generate a constant GND or VDD. This constant would then be used as an input to other primitives in the CLB, such as a flip-flop or a CARRY4.

This case is very similar to the previous case of a LUT routethru, in that the LUT is now playing an active role in the design, where there would have originally been no LUT to represent this constant in the original netlist. It should be noted that this is quite rare; usually Vivado will route a constant from a dedicated constant generator in the global routing fabric; however, on occasion it seems to decide to use a LUT instead, possibly when there is increased routing congestion.

We follow a similar approach to the LUT routethru case, where we add a new LUT6_2 instance to the design, and add a new net to connect from the LUT output to the required primitive input. The LUT INIT is set to all 0s or all 1s to generate the respective GND or VDD constant. Again, care must be taken when this new constant generator shares the LUT6_2 with another logic function in the netlist (or with a routethru).

6) LUTRAMs: The Xilinx UNISIM library has multiple primitives for LUTRAMs which use anywhere from one to four LUTs [23]. When analyzing the LUTRAMs, fasm2bels seeks to map to the largest LUTRAM primitive possible. This does not always match the synthesized netlist, since the synthesis tool may chose smaller LUTRAM primitives, but then in implementation, they will get packed together on the same slice, mimicking the behavior of a larger primitive. Therefore, we analyze the placement of the LUTRAMs in the design implementation and combine them into larger LUTRAM primitives in the produced netlist, where applicable.

B. Maintaining Functional Equivalence

The goal of our assurance flow is to verify functional equivalence back to the original post-synthesis netlist. Since the actual comparison step compares against the physical netlist (not the post-synthesis netlist), care must be taken to

ensure that none of the transformations made in the physical netlist generator change the functionality of the design.

Fortunately, for most of these transformations it is fairly obvious they do not impact the netlist functionality. The constant generator and LUT routethru modifications only serve to replace existing routing and constant generation with LUT6_2 primitives, and do not impact the functionality. Combining two LUTs into a single LUT6_2 also does not change their functionality, provided the INIT value is updated correctly. The same is true for any input pin reordering.

Perhaps the highest risk of change to functionality is when UNISIM cells are replaced by their physical implementation, as shown in Figure 2. Care must be taken to ensure parameters and signals are set correctly to match the corresponding primitive.

C. Trusting the Implementation Information

There may be some concern that we are using untrusted information from the Vivado tool to make these transformations. For example, if the tool provided false information about the implemented design, it could lead us to make incorrect transformations. However, since the transformations we make don't actually change the functionality of the design, there is no risk of this attack vector causing us to falsely verify the design. With incorrect implementation information, the design would simply be transformed incorrectly, and not exactly match the structure of the reverse-engineered netlist. This could cause false negatives (where equivalent bitstreams are reported as non-equivalent), but not false positives (where non-equivalent bitstreams are reported as equivalent).

V. Structural Mapping and Comparison

Once the physical netlist has been generated, we use Spydrnet [21] to create the netlist data structures for our mapping algorithm.

Our structural mapping algorithm is shown in Listing 1. The general approach for our algorithm is to iterate through all cell instances in the netlist design, and determine which cell instances in the reserve-engineered netlist could be possible matches. When the number of possible matches is one, a match is made. If the number of possible matches is zero, then the algorithm reports that the designs are not equivalent.

The algorithm begins with an empty mapping of nets and instances (line 2). It then loops through all cells in the netlist design and determines a list of possible matches in the reverse-engineered netlist, based on the cell type and cell properties. These possible matches are cached for each cell instance (lines 5-12).

The next part of the algorithm uses the top-level pin constraints to establish a mapping between the top-level nets in the netlist and the reverse-engineered netlist (lines 16-19). This is necessary because the reverse-engineered netlist does not contain any signal names, so we need to use the constraints to determine which top-level nets are connected to which pins. This also provides an initial seed mapping for the algorithm, which will help it make progress in the early stages of the algorithm.

Once these setup steps are complete, the algorithm enters the main phase of the matching (lines 22-40). For each instance, a list of possible matches are initialized to the cached list of possible matches (line 27), and then filtered to only those that have not yet been mapped (lines 28-29). The algorithm then filters this list based on the mapped nets connected to the cell pins (line 30-31). The more pins that are connected to mapped nets, the better the filtering will perform. If the number of possible matches is greater than one, then the

Listing 1: Structural mapping algorithm to map instances and nets between two netlists, ***A*** and ***B***

```
1  # Net, instance map are empty bi-direction maps
2  net_map = bidict(); block_map = bidict()
3
4  # Initialize a cache of possible matches
5  possible_matches = dict()
6  for A_inst in A.instances:
7      # Filter potential matches in B
8      matches = [m for m in B.instances
9          if m.type == A_inst.type]
10     matches = [m for m in matches
11         if props_match(m.properties, A_inst)]
12     possible_matches[A_inst] = matches
13
14 # Map nets connected to top-level pins.
15 # (These match based on design constraints)
16 for A_pin in A.top_pins:
17     net_A = A_pin.get_net()
18     net_B = B.get_pin(pin_A.name).get_net()
19     net_map[net_A] = net_B
20
21 # Loop until no more progress has been made
22 while progress:
23   progress = False
24
25   # Visit all unmapped instances in A
26   for A_inst in unmapped A.instances:
27     matches = possible_matches[A_inst]
28     matches = [m for m in matches
29         if m not in block_map]
30     matches = [m for m in matches
31         if pin_nets_match(m, A_inst, net_map)]
32     if len(matches) > 1:
33         possible_matches[A_inst] = matches
34         continue # Mapping ambiguous
35     if len(matches) == 0:
36         error_and_exit() # Mapping impossible
37
38     # Mapping is unique
39     add_instance_mapping(A_inst, matches[0])
40     progress = True
41
42 def add_instance_mapping(A_inSst, B_inst):
43     block_map[A_inst] = B_inst # Map instance
44
45     # Use new mapping to map nets
46     for pin_A in A_inst:
47     net_A = pin_A.get_net()
48     net_B = B_inst.get_pin(pin_A.name).get_net()
49     if net_B if net_map:
50     error_and_exit() # Net already mapped
51     if net_A not in net_map:
52     net_map[net_A] = net_B # Map net
```

TABLE I: Validated Designs

Design	Resources						Runtime (s)		# Error Injection Runs
	# LUTs	# FF	# CARRY4	# BRAM	# LUTRAM	# SRL	Phys. Netlist.	Struct. Cmp.	
stereovision1	13164	11588	2014	0	0	0	8.7	237.4	100
aes128	2790	4480	0	86	0	0	3.6	42.8	100
riscv_final	1499	1390	44	0	0	0	4.8	7.7	100
cpu8080	1010	243	86	0	0	0	1.8	2.7	100
sha	1000	894	56	0	0	0	1.5	3.4	100
mkSMAdapter4B	987	1126	73	4	0	0	1.1	4.8	100
bubblesort	814	1782	0	0	0	1	1.5	5.7	100
pid	741	423	0	0	0	0	1.2	3.3	100
median	740	125	52	0	0	0	1.2	3.1	100
a25_decode	677	640	0	0	0	0	0.7	5.0	100
regfile	611	1056	0	0	0	0	3.7	5.6	100
riscvSimpleDatapath	570	63	28	0	0	0	3.7	2.5	100
basicrsa	540	459	72	0	0	0	0.7	1.9	100
hight	502	134	28	0	0	0	0.7	1.3	100
alu	461	0	20	0	0	0	3.3	2.6	100
a25_wishbone	422	818	0	0	0	0	0.6	2.7	100
uart2spi	369	410	6	0	0	0	0.9	1.2	100
quadratic_func	238	118	52	0	0	0	0.4	1.4	100
raygentop	221	303	4	0	0	1	0.5	1.4	100
pci_mini	219	333	0	0	0	0	0.6	1.9	100
tiny_encryption_algorithm	200	264	40	0	0	0	0.5	2.4	100
data_path	179	257	3	0	0	0	0.5	1.6	100
EX_stage	168	38	4	0	0	0	0.4	0.9	100
calc	163	18	12	0	0	0	3.1	1.3	100
pic	133	77	8	0	0	0	0.4	0.4	100
wb_lcd	87	80	5	0	0	0	0.3	0.5	100
control_unit	78	5	0	0	0	0	0.3	0.4	100
a25_coprocessor	74	171	0	0	0	0	0.3	0.6	100
uart	69	137	18	0	0	0	3.0	1.5	100
stereovision3	54	118	0	0	0	0	0.1	0.4	100
shiftReg	51	20	0	0	0	0	3.0	1.1	100
UpDownButtonCount	49	24	12	0	0	0	3.0	1.3	100
simon_core	35	27	0	0	0	12	0.2	0.3	100
stopwatch	34	52	10	0	0	0	2.9	1.1	100
stereovision2	28	39	0	0	0	0	0.1	0.3	100
ID_stage	26	73	0	0	0	0	0.2	0.5	100
bcd_adder	24	50	5	0	0	0	0.3	0.2	100
uart_rx	20	39	4	0	0	0	0.1	0.2	100
rx	19	39	4	0	0	0	2.8	1.3	100
random_pulse_generator	4	33	0	0	0	0	0.2	0.1	100
a25_write_back	1	44	0	0	0	0	0.1	0.4	100
MEM_stage	0	37	0	0	64	0	0.2	0.3	100

algorithm updates the cached list of possible cells with the new reduced list, but skips mapping this instance for now, and will return to it later (lines 32-34). If the number of matches is zero, then the algorithm reports that the designs are not equivalent, and exits (lines 35-36).

When there is a single possible match, the algorithm has successfully establish a new instance mapping, and the *add_intance_mapping* function (lines 42-52) is called. This function adds the mapping to the instance map, and then uses this new mapping to iterate over all of the cell pins, and establish mappings for the nets connected to the pins. Enhancing the net map will make future instance filtering more effective, and will help the algorithm make more progress in subsequent iterations.

This process is repeated continuously, as long as progress can be made. We have yet to encounter a design where this algorithm does not converge to a solution. However, there is a possible case, where there remain no unmapped nets, and all the unmapped primitives are of the same type and have the same property values. In this situation, the remaining primitives can simply be arbitrarily mapped to each other since they are all equal, allowing the algorithm to continue to make progress.

The code in Listing 1 serves as the *mapping* phase of the comparison process, and once completed, a *verification* phase is run. This algorithm iterates through all instances in the map, and ensures they are identical to each other, which consists of again verifying matching properties, and verifying that for each pin, the equivalent (ie mapped) nets are connected. This final check verifies that the entire design is mapped, and that every instance and net is identical between the two designs.

Our algorithm is implemented in pure Python code, so the performance is not nearly as fast as what could be achieved with a compiled language. However, we have thus far been focused on demonstrating the feasibility of the approach, and have not yet focused on performance. As the physical

transformation process only makes one pass through each cell in the design, it scales linearly with the size of the design. As for the structural matching, the worst-case runtime for this algorithm scales quadratically with the design size. This is because each loop in the matching iterates through all cells in the design, and in the worst case, the loop executes once for each cell in the design (at least one cell needs to be mapped during each iteration for the algorithm to progress). In practice, the algorithm typically makes multiple matches per pass (at minimum, the first pass matches all I/Os). Based on the fact that the algorithm has always made progress and converged to a solution, we believe that the algorithm is fast enough to be used in practice, and that it will scale successfully to larger designs.

VI. Experimental Results

A. Validated Designs

Table I shows the designs that we have validated using our physical netlist generator and comparison tool. It also shows the runtimes for the physical netlist generation and structural comparison/validation in seconds. This was run using Python 3.11 on an Intel i9-13900K CPU with 32 GB of RAM. We have 42 designs where we were able to validate the bitstream. These designs are a mixture of open-source IP from Open Cores, the VTR7 [24] benchmark suite, IP from the LEON3 processor, and some small designs of our own creation. The target device used in the experiments is a Xilinx Artix 7 XC7A200T FPGA.

The largest design is stereovision1 with 13,164 LUTS and 11,588 flip flops, and it took just under four minutes to run. This design is an order of magnitude larger, and ran orders of magnitude faster, than the largest designs tested in [2], which used a commercial equivalence checker. Given we are using a pure Python implementation, we believe this is a very promising result, and shows the feasibility of this approach to handle even larger, real-world design sizes.

We also had 13 designs that failed comparison (not shown in the table). However, this was not due to issues with our approach, but rather due to limitations in the third-party tools we used. Five designs failed because of a bug in the Python netlist reader, Spydrnet [21], causing it to fail to read the netlist. One design failed when using Project X-RAY to convert bitstream to FASM, and the rest failed due to fasm2bels failing to convert the FASM file to a correct netlist. This is not surprising since Project X-Ray and fasm2bels do not support all primitives offered by the Xilinx 7-series FPGAs. In general, it supports LUTs and flip-flops, LUT RAMs, SRLs, clock generators, and limited support for BRAMs. Other primitives, such as FIFOs and DSPs are not supported, so we had to ensure our test designs contained only supported primitives, or when possible, disable the use of these primitives in Vivado. However, even when attempting to restrict the design to be supported by these tools, we still encountered failures.

B. False Positive Testing

To verify the ability of the structural comparison tool to detect instances where the design has been altered, we created an error injector tool that would modify the reversed netlist, either randomly changing a bit in a LUT INIT property or randomly swapping the driving wires of two internal instance pins. The structural matcher was then run as usual, and we would verify that it reported a failed comparison. Table I includes how many different erroneous design modifications were tested for each of our benchmarks. None of the errors injected resulted in false positives.

VII. Discussion

Although we have been very pleased with the results of our approach, there are still some limitations and challenges that we have encountered.

Device Support and Larger Designs: The netlist conversion techniques we employ need to account for every possible transformation that can occur during the post-synthesis implementation and bitstream generation phases of the CAD flow. Unfortunately, we are not privy to the Xilinx CAD tool internals, so there is no way for us to know whether our catalog of transformations is comprehensive. Although our approach is ad hoc, we have accounted for all transformations we have observed, and have purposefully included a very large number of test designs in our results to demonstrate our due diligence in discovering possible transformations. We have not withheld any designs from the results (except those that failed due to third party tool issues).

Bitstream documentation projects, like Project X-Ray[19], use differential analysis to determine how FPGA features map to the bitstream. However, obtaining complete coverage of the bitstream is challenging, and requires additional engineering beyond what has been invested so far into this open source project. When using other vendors, the designer would have to be aware of the physical transformations between the logical netlist and the physical implementation on the device. While we wanted to demonstrate our work on even larger designs, we found that the larger the design, the more likely we were to encounter an unsupported feature. Despite these challenges, we feel this approach would still be useful for a security-conscious organization that was willing to invest the time and effort to ensure the bitstream documentation was complete for their target device.

Trust Model: Our approach is designed to verify the correctness of the CAD tools, from netlist to bitstream. As such, our approach does not rely on trusting any information reported by the CAD tools.

We do leverage RapidWright [3], a partially open-source tool, to parse the Vivado implementation information and generate the new physical netlist. However, even without RapidWright, this same information could be retrieved via the Vivado Tcl API. As discussed earlier, if the retrieved implementation information was in some way false, it would not impact the correctness of our approach, but could lead to false negatives.

RapidWright is not actually capable of generating a Verilog netlist, so we use Vivado to convert the EDIF physical netlist generated by RapidWright to Verilog. This was done for simplicity, but would ideally be replaced by a trusted tool in a production assurance flow.

We leverage the open-source Spydrnet [21] netlist parser in our structural comparison process, but this tool is not required, and could be replaced with another tool, or even a custom parser.

Other Malicious Circuitry: Our structural comparison approach verifies the exact equivalence of all circuitry between inputs and outputs of the design. Currently we do not look for or analyze "floating" circuitry that may be present in the bitstream, such as power-generating ring oscillators, or short-circuits [25]. However, other research projects have demonstrated this type of analysis and detection [26].

VIII. Conclusion

We demonstrate a novel method to verify synthesized netlist-to-bitstream equivalence, proving the correctness of the implementation and bitstream generation steps of the CAD flow. The presented approach would also detect attacks that intercept and modify the bitstream, such as those demonstrated in [14]–[16].

We have provided an open-source release of these tools, as well as experimental results that show our structural comparison accurately detects errors in altered bitstreams.

Our method is a viable option to validate a bitstream and scales better than previous techniques at attempting to prove functional equivalence through formal methods.

References

[1] Trusted Microelectronics Joint Working Group, "New methods to instill trust in commercial semiconductor fabrication," White Paper, Jul. 2017.

[2] E. Cahill, B. Hutchings, and J. Goeders, "Approaches for FPGA design assurance," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 3, 28:1–28:29, Dec. 28, 2022.

[3] C. Lavin and A. Kaviani, "RapidWright: Enabling custom crafted implementations for FPGAs," in *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2018, pp. 133–140.

[4] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 1, 6:1–6:23, May 27, 2016.

[5] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[6] H. Salmani, M. Tehranipoor, and J. Plusquellic, "A novel technique for improving hardware trojan detection and reducing trojan activation time," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 1, pp. 112–125, Jan. 2012.

[7] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "VeriTrust: Verification for hardware trust," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, Jul. 2015.

[8] P. Kitsos, K. Stefanidis, and A. G. Voyiatzis, "TERO-based detection of hardware trojans on FPGA implementation of the AES algorithm," in *Euromicro Conference on Digital System Design (DSD)*, Aug. 2016, pp. 678–681.

[9] L. Pyrgas, F. Pirpilidis, A. Panayiotarou, and P. Kitsos, "Thermal sensor based hardware trojan detection in FPGAs," in *Euromicro Conference on Digital System Design (DSD)*, Aug. 2017, pp. 268–273.

[10] J. He, Y. Zhao, X. Guo, and Y. Jin, "Hardware trojan detection through chip-free electromagnetic side-channel statistical analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2939–2948, Oct. 2017.

[11] M. Lecomte, J. Fournier, and P. Maurine, "An on-chip technique to detect hardware trojans and assist counterfeit identification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 12, pp. 3317–3330, Dec. 2017.

[12] S. Narasimhan, D. Du, R. S. Chakraborty, *et al.*, "Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach," in *International Symposium on Hardware-Oriented Security and Trust (HOST)*, Jun. 2010, pp. 13–18.

[13] X. Zhang, A. Ferraiuolo, and M. Tehranipoor, "Detection of trojans using a combined ring oscillator network and off-chip transient power analysis," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 9, no. 3, 25:1–25:20, Oct. 8, 2013.

[14] C. Krieg, C. Wolf, A. Jantsch, and T. Zseby, "Toggle MUX: How x-optimism can lead to malicious hardware," in *Design Automation Conference (DAC)*, Jun. 2017, pp. 1–6.

[15] C. Krieg, C. Wolf, and A. Jantsch, "Malicious LUT: A stealthy FPGA trojan injected and triggered by the design flow," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[16] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA trojans through detecting and weakening of cryptographic primitives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1236–1249, Aug. 2015.

[17] A. Hastings, S. Jensen, J. Goeders, and B. Hutchings, "Using physical and functional comparisons to assure 3rd-party IP for modern FPGAs," in *International Verification and Security Workshop (IVSW)*, Jul. 2018, pp. 80–86.

[18] Yosys Open SYnthesis Suite. "Project IceStorm." (Oct. 28, 2021), [Online]. Available: https://github.com/YosysHQ/icestorm (visited on 10/31/2021).

[19] F4PGA. "F4pga/prjxray." (Jun. 23, 2020), [Online]. Available: https://github.com/f4pga/prjxray (visited on 06/23/2020).
[20] CHIPS Alliance. "Chipsalliance/f4pga-xc-fasm2bels." (Apr. 10, 2023), [Online]. Available: https://github.com/chipsalliance/f4pga-xc-fasm2bels (visited on 07/24/2023).
[21] BYU Configurable Computing Laboratory. "SpyDrNet." (Jun. 16, 2023), [Online]. Available: https://github.com/byuccl/spydrnet (visited on 07/27/2023).
[22] Xilinx Inc., *Vivado design suite 7 series FPGA libraries guide (UG953)*, 2012.
[23] Xilinx Inc., *7 series FPGAs configurable logic block user guide (UG474)*, 2016.
[24] J. Luu, J. Goeders, M. Wainberg, *et al.*, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 7, no. 2, pp. 1–30, Jun. 2014.
[25] H. Cook, J. Arscott, B. George, T. Gaskin, J. Goeders, and B. Hutchings, "Inducing non-uniform FPGA aging using configuration-based short circuits," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, 41:1–41:33, Jun. 6, 2022.
[26] T. M. La, K. Matas, N. Grunchevski, K. D. Pham, and D. Koch, "FPGADefender: Malicious self-oscillator scanning for xilinx UltraScale + FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 3, 15:1–15:31, Sep. 1, 2020.