Approaches for FPGA Design Assurance

ELI CAHILL, Brigham Young University, USA
BRAD HUTCHINGS, Brigham Young University, USA
JEFFREY GOEDERS, Brigham Young University, USA

Field-Programmable Gate Arrays (FPGAs) are widely used for custom hardware implementations, including in many security-sensitive industries, such as defense, communications, transportation, medical, and more. Compiling source hardware descriptions to FPGA bitstreams requires the use of complex computer-aided design (CAD) tools. These tools are typically proprietary and closed-source, and it is not possible to easily determine that the produced bitstream is equivalent to the source design.

In this work we present various FPGA design flows that leverage pre-synthesizing or pre-implementing parts of the design, combined with open-source synthesis tools, bitstream-to-netlist tools, and commercial equivalence checking tools, to verify that a produced hardware design is equivalent to the designer's source design.

We evaluate these different design flows on several benchmark circuits, and demonstrate that they are effective at detecting malicious modifications made to the design during compilation. We compare our proposed design flows with baseline commercial design flows and measure the overheads to area and runtime.

ACM Reference Format:
Eli Cahill, Brad Hutchings, and Jeffrey Goeders. 2021. Approaches for FPGA Design Assurance. *ACM Trans. Reconfig. Technol. Syst.* 1, 1 (October 2021), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Field-Programmable Gate Arrays (FPGAs) offer a "blank-slate" of programmable circuitry, allowing designers to implement arbitrary digital circuits that have the potential to provide speed and power improvements over commodity software processors, while providing faster time-to-market and lower designs costs than ASICs. This enormous level of flexibility has led to FPGAs being used in a diverse range of technology sectors, including several areas where design integrity is key, including defense applications, biomedical devices, self-driving vehicles, communications infrastructure, and more.

However, this great flexibility of FPGAs comes at a price: complex CAD tools are required to map the designer-provided circuit description to a *bitstream* that can be used to configure the FPGA. When used in a standard out-of-the-box configuration, these commercial CAD tools perform many transformations and make it nearly impossible for a designer to easily

Authors' addresses: Eli Cahill, , Brigham Young University, EB 450, Provo, UT, USA; Brad Hutchings, brad_hutchings@byu.edu, Brigham Young University, EB 450, Provo, UT, USA; Jeffrey Goeders, jgoeders@byu.edu, Brigham Young University, EB 450, Provo, UT, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Association for Computing Machinery.
1936-7406/2021/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

verify on their own that the produced bitstream precisely implements their original design, without unintended data leaks, back doors, or other compromising features. However, as we show in this paper, by combining commercial tools with recent advances in open-source FPGA tools and equivalence checking tools, it is possible to perform design equivalence checking. However, there remains significant limitations on synthesis optimizations, scalability to larger designs, and portability to new FPGA families.

1.1 Motivation and Objectives

The FPGA CAD flow consists of many stages and connected tools as the design is transformed step-by-step from a textual behavioral description, to a physical programming file. In general, these tools are closed-source, and the produced bitstream is of propriety format, leaving designers to trust that their designs are implemented on the FPGA accurately and without compromising features.

While the software world offers many tools to inspect the compilation process and end-result executable (open-source tools, dissasemblers, dynamic monitors), the compilation tools used in the FPGA CAD process are almost exclusively close-sourced, proprietary tools. These tools, which are typically tens of gigabytes in size and can take hours to compile designs, are nearly black-box systems. In addition, the produced configuration bitstream is proprietary, and the designer has no way of knowing what is actually implemented in the circuitry specified by the bitstream. The designer is forced to completely trust that the CAD tools implemented the design as requested. Unfortunately this leaves several vulnerabilities in the FPGA compilation process:

- A malicious actor within the company (or contracted company) responsible for the CAD tools may be able to modify the tool to silently inject back doors, kill switches, or other hardware Trojans into the design.
- An unknown bug in the complex CAD tools may produce a design where certain internal signals, such as encryption keys, or other sensitive data, are leaked and unintentionally observable to the larger system.
- Targeted 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.
- The generated bitstream could be intercepted and modified post-compilation, without the designer realizing it was changed after generation by the CAD tool. Again, this could be done by compromised tools, or it could by modified on the filesystem by an internal malicious actor, during file transfer, via network intrusion, etc.

As expected, these vulnerabilities may be of great concern for many designers using FPGAs. For example, defense contractors may be concerned with detecting hidden kill switches, communications companies may be unsure whether their design contains back doors that allow a foreign state to monitor communications, producers of medical devices may wonder whether patient confidentiality is sufficiently maintained, and designers of self-driving cars would be concerned with any design modifications that could compromise the safety of the passengers.

While such scenarios may seem unlikely, there have been some cases of suspected hardware Trojans in the wild [1], [2], as well as several academic works that have demonstrated the feasibility of inserting hardware Trojans [3]–[5]. In fact, much research goes into ensuring that the source description of designs are Trojan free, before being synthesized into actual

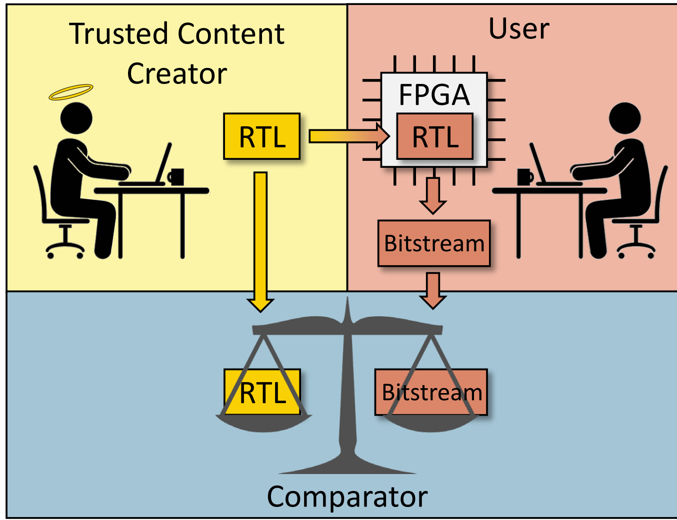


Fig. 1. Extracting and comparing circuit properties

hardware. It therefore makes sense to expend some effort to ensure that these designs remain safe and secure through the compilation process.

The end-goal of this work is to develop techniques to determine equivalence between the original HDL circuit description, and the low-level circuit produced by the closed-source FPGA CAD tools. The overarching approach taken in this project is illustrated in Figure 1, and consists of extracting properties and circuit details from the user-supplied design, and comparing them against properties extracted from the final FPGA bitstream.

1.2 Challenges and Strategies

While ideally one could simply provide the original design and the bitstream to a commercial equivalence checking tool, this is not feasible. Developing a full comparison flow, and associated tools, capable of determining equivalence for *arbitrary* circuits is a monumental task, due to several challenges:

- The bitstream is propriety, so cannot directly be input to commercial equivalence checking tools. In addition, even if the bitstream can be converted to an equivalent netlist, all signal names from the original design are lost, making equivalence checking much more challenging.
- During the CAD flow, the design undergoes several transformations that make comparison difficult. For example, we created a simple 32-bit counter design that used the most-significant bit to blink an LED. The FPGA CAD tools were able to optimize away a single flip-flop, causing even this simple design to fail equivalence checking using Cadence Conformal (an ASIC-targeted formal verification tool).
- Modern equivalence checking tools are limited in scalability; they cannot tackle modern FPGA designs which can contain millions of logic gates.

Given these challenges, this work represents only the first steps towards the goal of arbitrary FPGA design equivalence checking, and leverages some of the following concessions to make the comparison process more manageable:

Disabling CAD Optimizations Certain optimizations can be disabled in the CAD flow to reduce the modifications made to the design during compilation. This includes disabling optimizations such as retiming, or applying code pragmas to prevent certain signals from being optimized away. We expect this would reduce the final quality of result, but will make equivalence checking easier.

Leveraging Open-Source CAD Tools Some parts of the CAD process (e.g. synthesis) can be performed on the source design prior to entering the commercial CAD flow. If trusted open-source tools are used, then it may be sufficient to compare the design against this preprocessed design (e.g. a pre-synthesized and optimized netlist), rather than the original RTL.

Only Check Certain IP One alternative is to perform comparison on only a portion of the design, such as certain sensitive IP modules like an encryption block. This reduces the size of the circuit that needs to be compared, and makes it feasible to use a commercial formal verification tool.

Pre-Bitstream Comparison Rather than comparing against the final bitstream, one can instead compare against the final netlist reported by the commercial CAD tool. While this means we must trust the CAD tool to report the netlist accurately, and it isn't useful to detect post-generation modifications to the bitstream, it is still a step in the right direction.

Given these different approaches to the problem, it is possible to explore many different FPGA compilation and comparison flows. In this article we present a few different approaches we took that can successfully determine equivalence between source design and compiled circuit, and can successfully detect malicious modifications to the design during synthesis and implementation. Each of these approaches requires a non-standard FPGA CAD flow, and we evaluate the QoR of the produced design against a standard commercial flow.

1.3 Organization

Our different approaches are summarized in this subsection, and then detailed and evaluated in subsequent sections of the paper (Sections 3 to 5). Each of these sections discusses the details of the proposed CAD flow, and evaluates it in terms of QoR penalty versus a traditional commercial CAD flow. Experiments are also performed to inject malicious design modifications to ensure our proposed flows always detect the modification. Section 6 then provides a comparison of the different design flows, discussing their respective strengths and limitations. Section 7 discusses conclusions and future work.

1.3.1 IP-Level Physical Assurance (Section 3) In our first explored CAD flow, we limited the scope to verifying equivalence for a single IP module in a design. While ideally one would be able to verify equivalence for an entire design, there are still many cases where just determining equivalence for a single IP module would be desirable. In modern FPGA design, 3rd-party Intellectual Property (IP) is often used to reduce the time and cost of the design process. However, the use of 3rd-party IP is not without risk: the inherent complexity of most 3rd-party IP modules makes it difficult for the user to determine whether or not the IP contains anything malicious, such as hardware Trojans.

One step towards securing 3rd-party IP would be to use IP only from trusted 3rd-parties who have verified and can vouch for the safety of the IP. However, even in this case, one may still be concerned with how the IP is implemented by the CAD tools, and whether it remains secure through the CAD process. This is the problem we address in this compilation and comparison flow.

In this proposed flow, which we refer to as a *Physical IP Assurance* flow, we assume the IP module is provided as a placed and routed physical partition that can be instanced in a user design. This places the burden on the IP provider to ensure the placed and routed design is safe. However, the comparison process is relatively simple as we can ensure that every configuration element remains identical to the original design. This may seem relatively uninteresting and straightforward, but it serves as a good starting point, and even in this simple case the CAD tool can make some unexpected optimizations that break equivalency. This work was first published in [6].

1.3.2 IP-Level Functional Assurance (Section 4) The next CAD flow we explore also targets a comparison process for individual IP modules. In this case, the source IP is a netlist that has already undergone synthesis and logic optimizations. The IP is then instantiated in a user design, with constraints added to ensure the CAD flow does not modify this IP.

In this case, determining equivalency is done using a commercial tool, Cadence Conformal, and some additional considerations are needed to deal with optimizations performed by the CAD flow. This approach was also previously published in [6].

1.3.3 Bitstream-Level Assurance (Section 5) The previously mentioned techniques leverage the CAD tool to request details of the implemented design. If the CAD tool were compromised, or if the generated bitstream were modified post-generation, these approaches may not be effective. In these cases, it is necessary to understand the contents of the produced bitstream, and compare it directly to the original circuit design.

In this section of our work, we use relatively new open-source tools that are capable of converting a proprietary FPGA bitstream to netlist form. This is done using Project Icestorm [7], which is able to generate technology-mapped netlist files from Lattice iCE40 FPGA bitstream files. This bitstream-reversed netlist is then compared against the original design, again leveraging commercial equivalence checking tools.

However, the comparison process is much more challenging at this point, as the reverse-engineered netlist has no signal names from the original design. This also defeats the simplification taken in our earlier work of just focusing on individual IP in the design, as it is very challenging to extract individual IP logic out of a netlist with no signal names. Instead, we are forced to perform comparison on the entire design.

Due to these challenges, our initial attempts at comparison failed, and we instead had to explore different CAD flows that would make comparison possible. To accomplish this, we created a Python framework called the BYU FPGA Assurance Tools (`bfasst`), which allows for easily composing different CAD flows combining different synthesis, implementation, bitstream reversal, and formal verification tools, in order to evaluate different design flows on a large set of benchmarks. To date, our experiments have shown we can perform successful verification, assuming we use open source synthesis tools to pre-optimize the design, as well as using modern verification tools targeted toward FPGAs [8].

2 Background

2.1 Related Work

2.1.1 Trojan Detection As the size, complexity, and use of digital circuits continues to grow, concern is rising about the presence of undetected *hardware Trojans* [2]. 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 deployed systems is limited [1], [2],

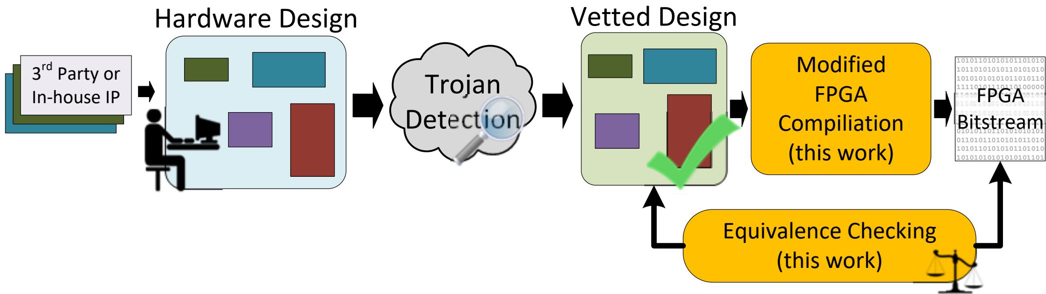


Fig. 2. FPGA compilation flow, demonstrating how the proposed fits together with existing work on Trojan detection.

their potential for severe destructive capabilities has motivated much research into techniques to detecting them.

Most of these works rely upon a circuit behaving as normal, then entering some externally triggered activation state. Techniques have investigated characterizing and detecting activation logic [9], [10], using embedded circuitry to measure power and temperature fluctuations [11]–[13], using external measurements such as monitoring supply voltage or electromagnetic effects [11], [14], [15], or a combination of these techniques [16].

In general, these Trojan detection techniques fall into two categories: 1) those techniques that inspect and analyze the netlist for suspected Trojans before synthesis [9], [10], and 2) those that operate in real-time to monitor the active system [11]–[16].

In the case of the former, these netlist analysis techniques are essential and can work in cooperation with our techniques. That is, an organization would first use these published techniques to determine that the hardware source code is free of Trojans, regardless of whether that code was created in-house, or as part of third-part IP. If the IP was obtained from a trusted party, it may not need to be vetted. Then, after the design is compiled to an FPGA bitstream, the techniques proposed in this project could be leveraged to gain assurance that the final design remains safe and untampered. This strategy is illustrated in Figure 2.

Without the techniques discussed in this work, one could employ the real-time detection techniques proposed in these other works; however, this may not always be adequate. Firstly, it may not be possible to deploy the necessary monitoring equipment on-site, and in every instance of the system. Secondly, while *detecting* an attack is helpful, the attack could still render the system inoperable, leak sensitive encryption data, etc.

2.1.2 Attacking FPGAs Our 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, these could occur with compromised CAD tools, or if the FPGA bitstream was compromised after generation. While these cases my seem improbable, recent work has shown both their possibility, and potential for significant consequences. In [3] and [4] 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. In [5] 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.

Recent work has even shown techniques to recover a fully encrypted bitstream [17].

2.1.3 Open-Source Bitstream Tools The bitstream formats used by the commercial FPGA CAD tools are proprietary, and it is not documented how changes to the bitstream affect the circuit implemented on the FPGA. In a sense, this obfuscation provides some protection, and makes it difficult to modify the bitstream to perform specific tasks (although not impossible, as demonstrated in [5]).

However, recent projects by many research groups have successfully documented the format of several FPGA bitstreams [18]. The ability to arbitrarily modify the FPGA bitstream allows for more advanced bitstream-level attacks, such as those that insert short circuits or glitches to draw huge amounts of power. This can have extreme effects including shutting down the system, accelerating circuit wear-out, broadcasting secret messages, or even melting the solder and permanently damaging the system [19]–[22].

However, the recent availability of FPGA bitstream-level analysis tools is also what enables this work to take place. Five years ago, the work presented here would not have been feasible. However, with current open-source tools such as *Project X-Ray* [23], *Project U-Ray* [24] and *Project Icestorm* [7], it is now possible to develop tools and techniques to determine equivalence between hardware designs and the resulting bitstreams.

2.1.4 Other Approaches for FPGA Design Assurance Several works have leveraged FPGA bitstream analysis to detect potentially malicious elements [25], [26], or to establish isolation between design elements [27]. However, these works are designed to detect specific patterns in the bitstream circuit. To our knowledge, there are no previously published tools or studies that attempt to explore general-purpose equivalence checking for the FPGA compilation process.

2.2 Threat Model

The FPGA assurance flows presented in this paper involve three parties, 1) a trusted content creator, 2) an end-user, and 3) a comparator as shown in Figure 1. Depending on the process, these three roles may be played by the same individual, or may be three separate entities.

Trusted Content Creator: The trusted content creator is responsible for creating and vetting the source design. In assurance flows where we target individual IP (Sections 3 and 4), the content creator is the provider of the IP library. In flows that seek to determine equivalence of the entire design (Section 5), the content creator becomes both the provider of any third party IP, as well as the designer that is integrating these IP with their own content into a final combined design.

We assume the content creator has sufficiently inspected and tested the source design to determine that it is free from malicious inclusions before beginning compilation. For example, they may employ the hardware Trojan detection and mitigation techniques described previously [9], [10].

User: The user represents the designer(s) that are compiling the final hardware design, and are interested in determining that the compiled design is equivalent to the original source hardware. In cases where the user is separate from the content creator, such as when a user is implementing trusted IP in their design, the user is not required to know hardware Trojan detection and mitigation techniques, nor understand the details within the IP.

Comparator: The trusted IP files and the user's final bitstream (or placed and routed design for pre-bitstream processes) are provided to a party that performs the comparison. The comparison between the trusted source hardware and the implementation of that hardware in the compiled design is completely automated.

The party performing the comparison process could be the original content provider, the user, or another trusted third party. There are a couple benefits to using a third party to perform the comparison process. First, it mitigates scenarios where an organization was concerned that a user could intentionally insert malicious hardware during compilation or post bitstream generation. Second, the techniques presented in this paper are meant to prevent attacks that could occur if the CAD tools on a user's system were compromised. In such a case, it may be possible for the attackers to also compromise the equivalence checking tools, making them falsely report equivalence. Having a separate party perform the comparison makes it substantially more difficult for an attacker, as they would have to compromise tools on two different systems, perhaps belonging to different networks and/or organizations.

3 An Approach for Physical-Level IP Assurance

This section discusses the first CAD flow we explored, which focuses on solving a narrow problem: verifying design equivalence for a single trusted source IP when that IP is already a pre-placed and pre-routed design partition. We explore the design flow where a user instantiates this IP in their design, and investigate whether the physical placement and routing remains completely untouched in the final implemented design.

While this may seem relatively uninteresting and straightforward, it serves as a good starting point, and interestingly, even when this high degree of restriction is placed on the source design, the CAD flow transformations can still break equivalency.

3.1 Physical Assurance Process

The detailed steps of the process are outlined below and are illustrated in Figure 3.

3.1.1 Trusted IP

(1) Trusted third party creates a Vivado project for a specific FPGA part using the Xilinx Vivado Hierarchical Design (HD) flow. Vivado HD contains several different flows centered around the idea of a partitioned design. In Xilinx terminology, a *Pblock* refers to a physical partition of the chip, containing a set number of resources.
(2) The trusted third party creates a Pblock, assigns their IP to this block, and performs synthesis and implementation. It is important to recognize that this will be *out-of-context* synthesis, meaning that the IP is synthesized independent from, and without any knowledge of the user circuit in which it will be instanced. This unfortunately prevents the synthesis tool from performing any cross-boundary optimizations. For example, if the user circuit did not use certain output ports of the IP, the synthesis tool could normally remove the logic from the IP that drives these ports; however, without this knowledge, such an optimization would not be possible.
(3) Once implemented, the IP is fully contained within this Pblock partition and can be exported as a Vivado Design Checkpoint (.dcp file). This file is then provided to the user.

A major disadvantage of this approach is that the IP provider must choose the FPGA part and exact Pblock location on the chip. This can be mitigated by providing multiple files

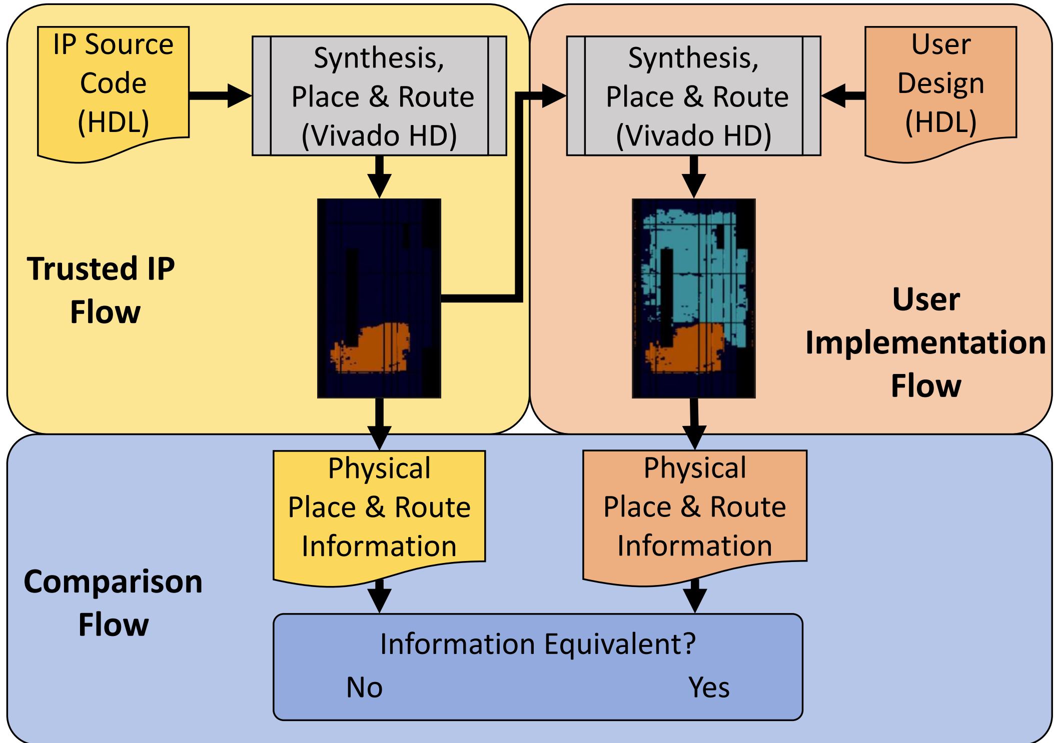


Fig. 3. Physical Assurance Flow

for different parts, and a few different locations on the chip to give greater flexibility to the user, although this places a larger burden on the content creator.

3.1.2 User Implementation

(1) The user creates their full design in HDL and synthesizes it using Vivado.
(2) The user creates a Pblock in their design in the same location as the one chosen by the trusted IP provider.
(3) The trusted IP (.dcp file) is assigned to the Pblock.
(4) The user design is implemented using Vivado, utilizing the IP from the trusted vendor.

3.1.3 Physical Comparison The final phase of the process is the comparison flow. The user uses Vivado to produce a checkpoint file (.dcp file) of the final placed and routed full design. This checkpoint, and the trusted IP checkpoint are both provided to the comparator party. The comparator party then runs a series of Tcl commands in Vivado to extract design properties about both design checkpoints. For the full design, the only design properties that are extracted are those from within the IP partition Pblock.

We have fully automated the extraction and comparison process using TCL scripts developed for this work. We assume the design tools accurately extract the trusted IP circuit elements. In the event that the reported results were spoofed, it would be necessary to instead extract design properties from the bitstream itself, bypassing any reporting from the untrusted CAD tool.

The extracted physical information used in the comparison process includes everything necessary to fully reconstruct the design on the fabric. In detail, the extracted information includes:

- Sites: a list of all of the physical resources allocated to the Pblock.
- Cells: the name, primitive type, configuration (LUT programming bits, for example), location, and a mapping of netlist pins to physical pins.
- Netlist, boundary nets: the names and pin locations of the nets connecting the IP to the top-level logic.
- Netlist, internal nets: the name, wires (if applicable), routing switch configuration (if applicable), and pin count, of nets internal to the IP.

The comparator party then compares these two sets of properties for equivalence. The goal of physical assurance is to compare the trusted IP and instantiated IP based on physical implementation details. If two designs use the same physical resources in the same way, the two designs are guaranteed to be (barring slight process variations) functionally identical. If a difference is detected, it would indicate that tampering has occurred during the compilation process. In reality, additional considerations need to be taken to remove false positives.

3.2 Challenges and Limitations of Physical Assurance

Physical assurance, as a research approach, was selected as it provides a strict guarantee that the IP is implemented in the exact same manner as provided by the trusted vendor, down to the individual wiring of each net. Although this introduces some overheads, the prevailing notion was that, if functional assurance became impossible for some reason, physical assurance could serve as a final assurance back-stop that would work in most, if not all, cases.

As an idea, physical assurance is simple enough. However, in practice, due to inherent CAD tool complexity, direct comparison of the physical manifestations of the trusted and instantiated IP often failed, and required workarounds. Some challenges we encountered were:

Global Resources: Locking down the placement and routing of the trusted IP block sometimes leads to problems. Circuits would fail to route if the trusted IP contained a global resource such as a clock buffer, as many global resources cannot be included in pBlock partitions. To avoid this problem, the IP creator would be prevented from instantiating global resources in their pBlock, and would instead have to instruct the user to instantiate the resource in their surrounding design.

Pblock location: In certain cases routing would fail likely because the location of the Pblock increased congestion. This problem can be mitigated if the provider of the trusted IP generates several versions of the trusted IP, each with different Pblock locations.

Subtle physical optimizations: Despite the fact that Pblock partitioning is meant to prevent any changes, in some cases, Vivado would occasionally permute the inputs at the periphery of the trusted IP Pblock. This problem caused most of the benchmarks to initially fail an equivalence test. This problem was overcome by extracting the LUT pin mapping from the original trusted IP, and then locking these pins prior to routing the instantiated IP.

Finally, in one synthetic benchmark containing a pipelined version of the MD5 algorithm (md5_pipelined), Vivado performed a minor optimization, (again, near the periphery of the Pblock) that occurred because two nets were aliases of one another. After manually inspecting the design, it was determined that the trusted IP and the instantiated IP were indeed functionally equivalent; however, they were not physically equivalent. After some

experimentation, we found that this minor physical difference could be eliminated if the placement of the Pblock was modified. This is somewhat similar to the problem where Pblock placement interfered with placement and routing (though movement of the Pblock as a solution is probably not the best solution). This problem only occurred in one synthetic benchmark.

3.3 Experiments with Physical Assurance

Experiments were conducted to answer the following questions for our physical assurance techniques:

- Can the assurance approach successfully extract the IP from the user's implemented design, compare it against the original trusted IP, and determine that they are identical?
- Can the assurance approach successfully detect modifications to the trusted IP that may have occurred anywhere in the insertion and implementation process?
- How does this assurance approach impact the timing constraints and area of the final implemented design? Or in other words, what does the end user have to "pay", in terms of speed and area, to achieve assurance using this technique?

3.3.1 Benchmark Designs Our experiments consist of several designs containing multiple IP that are each in turn treated as a "trusted IP". In total, 53 different IP modules were tested; these 53 IP were part of 22 different benchmark circuits. Of the 22, 21 are synthetic designs, created by interconnecting various modules from Open-Cores (www.opencores.org) without regard for circuit function. These designs represent a diverse set of module size, functionality and HDL type. The synthetic designs contain 43 of the IP of interest, as well as other hardware modules from Open-Cores that weren't tested, but serve as surrounding logic. The 22nd benchmark circuit is a functional LEON3 processor (www.gaisler.com/leon3) from which 10 of the modules instantiated at the top-level are included in our trusted IP set. In cases where benchmarks contained multiple trusted IP, multiple experiments were performed, each time selecting one IP module to play the role of the trusted IP block.

The extracted sub-blocks serving as the trusted IP ranged in size from tens to tens of thousands of logic cells. The benchmark statistics are provided in Table 1, and the full benchmarks are available at GitHub (www.github.com/byuccl/ipassurance). To set a baseline for timing and slice utilization, all benchmarks were implemented using a conventional synthesis, place, and route flow to determine whether timing constraints are met and to measure slice utilization. These baseline results are used to determine how much (if, at all) the Pblock-based assurance approach may affect timing and area.

Table 1. Benchmarks

Top-Level Benchmark	IP Module	LUTs	FFs	RAMB18	DSP48
leon3mp	(top)	33576	15963	124	16
	dsu3	517	407	4	0
	irqmp	504	227	0	0
	mctrl	207	150	0	0
	leon3s	7100	3260	28	4
	ahbuart	276	179	0	0
	apbctrl	126	89	0	0
	spimctrl	197	151	0	0
	ahbctrl	353	53	0	0
	ahbjtag	242	194	0	0

	grethm	2125	1145	6	0
Synthetic1	(top)	9391	7661	440	0
	aes128	3583	3533	344	0
Synthetic2	(top)	6575	2792	28	0
	amber	6011	2333	28	0
	basicrsa	540	459	0	0
Synthetic3	(top)	2202	2085	0	0
	atahost	451	331	0	0
Synthetic4	(top)	1088	1143	0	0
	bcd_adder	18	41	0	0
	big_counter	581	201	0	0
	bubblesort	402	901	0	0
Synthetic5	(top)	11	42	0	0
	counter	1	33	0	0
Synthetic6	(top)	2287	1818	0	0
	des3_area	535	64	0	0
Synthetic7	(top)	5426	3992	0	0
	des3_perf	5413	3992	0	0
Synthetic8	(top)	9545	6358	98	0
	dfadd	3759	2230	2	0
Synthetic9	(top)	3230	1071	0	43
	cpu8080	1020	244	0	0
	fixed_point_sqrt	460	32	0	24
	graphiti	1368	689	0	19
	hight	372	102	0	0
	lfsr_randgen	2	4	0	0
Synthetic10	(top)	7471	6574	7	14
	fm_3d_core	2867	2235	0	10
Synthetic11	(top)	35188	32686	2	240
	jpegencode	35187	32686	2	240
Synthetic12	(top)	21764	13447	20	13
	m32632	11270	3138	20	13
	md5_pipelined	9817	10176	0	0
	median	639	125	0	0
Synthetic13	(top)	12713	15202	49	20
	mpeg2fpga	10448	14314	43	20
	msp430_vhdl	1387	298	0	0
	natalius_8bit_risc	29	25	2	0
	neo430	816	565	4	0
Synthetic14	(top)	3235	2239	4	2
	pci_mini	216	291	0	0
	pic	248	114	0	0
	potato	2400	1665	4	0
	pwm	228	145	0	0
	quadratic_func	109	24	0	2
Synthetic15	(top)	6775	3176	0	0
	pid	806	385	0	0
Synthetic16	(top)	899	390	0	0

	pid_simple	886	390	0	0
Synthetic17	(top)	8191	4335	0	0
	random_pulse_generator	4	33	0	0
	sap	68	44	0	0
	sha3_high_throughput	5258	2144	0	0
	sha3_low_throughput	2639	2084	0	0
	simon_core	48	27	0	0
Synthetic18	(top)	32642	1598	0	0
	sudoku	32285	1367	0	0
	tiny_encryption_algorithm	198	231	0	0
Synthetic19	(top)	1532	1285	0	2
	uart2spi	483	414	0	0
Synthetic20	(top)	1811	1063	9	5
	vga	632	508	1	0
Synthetic21	(top)	311	219	0	0
	wb_lcd	107	79	0	0

3.3.2 Equivalence of Unmodified Circuits After improving our compilation flow to address the issues discussed in Section 3.2, every benchmark/trusted IP in our suite could be run through our physical assurance flow (Figure 3) and result in a perfect match between the trusted IP and the instantiated IP. In other words, our technique to detect circuit modifications resulted in no false positives.

3.3.3 Detecting Circuit Modifications In addition to verifying that the original circuits could be verified to be equivalent, we also wanted to ensure that any small modifications made to the trusted IP would cause the equivalence checking to fail; or in other words, ensuring there were no false negatives. Our automated tests performed each of the following modifications three times for each trusted IP:

- Randomly select a cell from the instantiating circuit and move it into the trusted IP Pblock.
- Randomly select a cell from the trusted IP and change its location within the Pblock.
- Randomly select a LUT or FF from the trusted IP and change its initialization equation/value.
- Randomly select a net, change the route it takes from its source to its sink.

These tests were chosen to mimic any change that might be made to the circuit, malicious or otherwise. Every modification that was performed was caught by the assurance process, verifying that the proposed physical assurance flow is extremely sensitive to tampering.

3.4 Impact of Physical Assurance on QoR

For each "trusted IP" and parent benchmark listed in Table 1, we performed compilation using the previously described physical assurance flow. We then compared the slice utilization against a standard compilation of the benchmark. All designs were targeted a Xilinx Artix-7 100T FPGA. The results can be seen in Figure 4. In most cases, the additional restrictions imposed by the physical assurance flow resulted in an increase in the total number of slices used. On average, slice usage increased by 4.7%. This penalty is expected, as when the IP is synthesized out-of-context, it cannot employ any cross boundary optimizations, such as constant propagation or unused logic removal. In some cases the slice count decreased. While

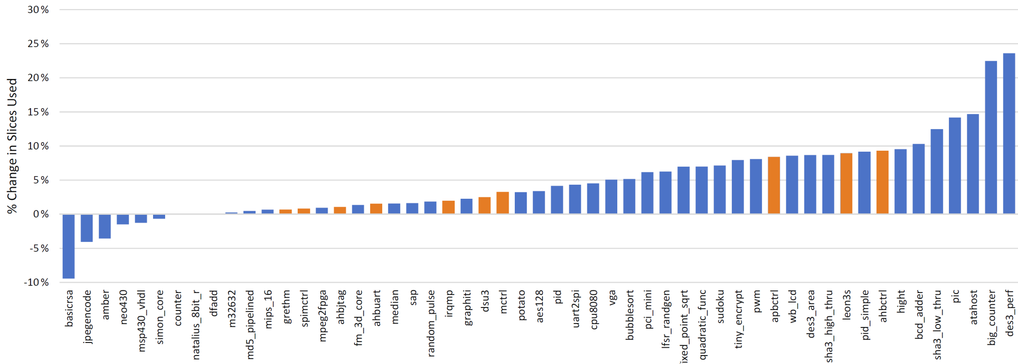


Fig. 4. Impact of physical assurance flow on slice utilization. Synthetic IP benchmarks (blue) and LEON3 IP benchmarks (orange).

somewhat unexpected, this is not altogether surprising; the stochastic nature of CAD tools will naturally produce circuits of varying sizes across different configurations, regardless of our assurance-based approach. In addition, changes in the circuit can affect the clustering algorithm and change how many logic elements are placed into each slice, meaning that while overall slice count could decrease, the internal slice utilization is higher.

We also explored the impact on timing results. Of the 10 experiments run on the LEON3 benchmark, all 10 passed the standard LEON3 timing constraints in both the baseline and Pblock-based designs. For the synthetic benchmarks, the timing results are less meaningful, although to gain some insight we applied a 10ns period constraint to all the synthetic designs. Of the 43 experiments created from the synthetic benchmarks, only one Pblock-based design (dfadd) failed timing after its baseline design had passed timing.

This suggests that while in some cases the Pblock approach will prevent the CAD tools from meeting timing, in many cases the timing is not significantly impacted.

4 An Approach for Functional-Level IP Assurance

This section describes the next major assurance process we explored, which we refer to as *Functional IP Assurance*. Like the previously described flow, this approach also is targeted to verifying individual IP instantiation within a larger design.

While the physical assurance process described in the previous section is effective, it places a large burden on the IP creator, requiring vetted placed and routed instances of the IP for any FPGA part the user wishes to use. In contrast, our functional assurance process described in this section aims to compare the IP at a *functional* level, ensuring that the netlist behavior is equivalent between the original IP specification and the final implemented design.

To determine equivalency between netlists, we leverage a commercial tool, Cadence Conformal. Conformal works by trying to map key points (registers) between the two netlists, and then verifies that the combinational logic between these registers is logically equivalent. This approach alleviates much of the burden placed on the IP provider by the physical assurance flow, as it no longer means the IP must be placed and routed for a specific part. Instead, the trusted IP can be specified as a netlist.

While ideally this would mean that the trusted IP could be provided simply as RTL code, in our testing, we found that the optimizations performed by the CAD flow during

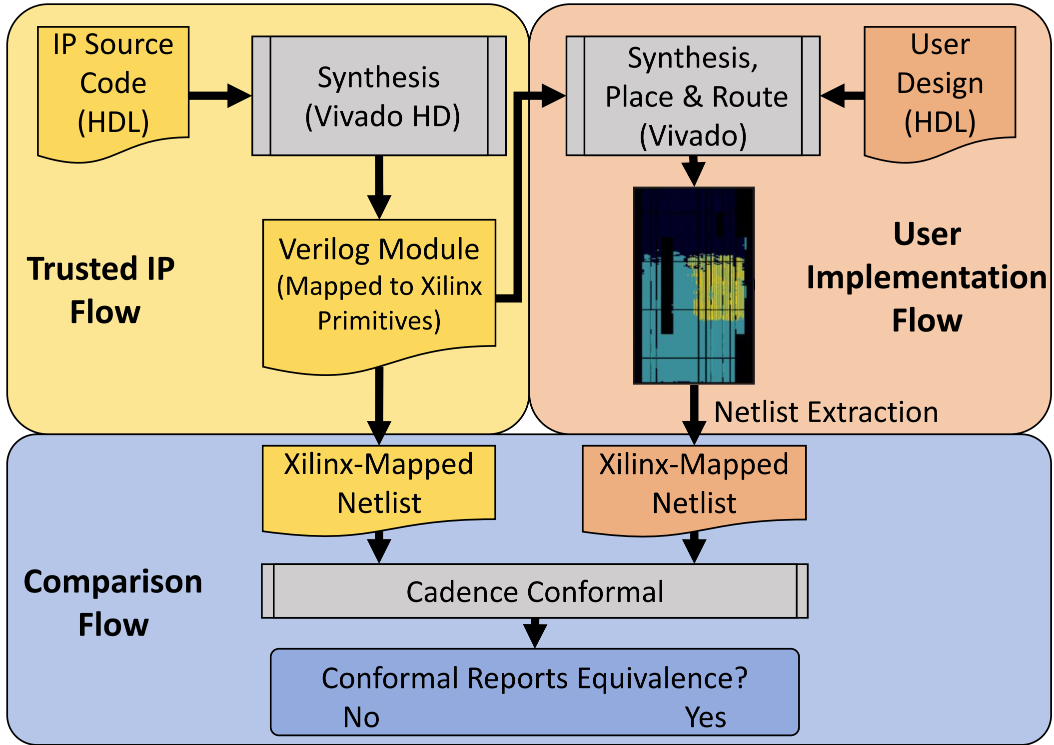


Fig. 5. Functional Assurance Flow

synthesis are too significant, and Conformal will almost always report that the designs were non-equivalent. Instead, we take the approach of pre-synthesizing the trusted IP, and then having the user incorporate the technology-mapped, optimized netlist into their full design. The full process is described next.

4.1 Functional Assurance Process

The functional assurance process is described below and is illustrated in Figure 5.

4.1.1 Trusted IP In the trusted IP flow, the trusted content creator creates the IP as a module using Vivado Hierarchical Design. Using the standard out-of-context CAD flow, the IP is synthesized, optimized, and mapped to Xilinx FPGA primitives. Finally, the trusted content creator extracts a Verilog netlist for their IP using the `write_verilog` command, which creates a netlist of Xilinx primitives (LUTs, FFs, etc.) This technology mapped netlist would then need to be vetted by the IP creator before being provided to the user.

4.1.2 User Implementation Using the standard Vivado flow, the user incorporates the netlist into their design. Because the trusted IP is in the form of a Verilog netlist, instantiating the IP is a straightforward and simple task; the user can instantiate the Verilog module in their RTL design in the same manner as any other Verilog module. In addition to instantiating the IP, the user must also apply a few additional constraints to prevent the CAD design tools from making further optimizations to the IP. This step makes it possible for the comparison

flow to determine equivalence between the trusted and instantiated IPs. We accomplished this step by applying the DONT_TOUCH attribute to the following objects:

- The cells within the instantiated IP
- The nets within the instantiated IP
- The hierarchical cell that represents the instantiated IP

For example, a user could apply these attributes to an IP named aes128_0 by adding the following lines of code to their constraints file:

Listing 1. Tcl commands to prevent further netlist optimization

```
set_property DONT_TOUCH true [get_cells aes128_0/*]
set_property DONT_TOUCH true [get_nets  aes128_0/*]
set_property DONT_TOUCH true [get_cells aes128_0]
```

The DONT_TOUCH attribute prevents the Xilinx CAD tools from optimizing or changing anything it is applied to. This forces the CAD tools to place and route the IP netlist without changing its structure, and therefore its behavior. While this initially may seem drastic, it is important to recognize that the trusted IP has already undergone logic optimization in the trusted IP Flow. The DONT_TOUCH attribute simply prevents *further* optimizations to the IP, such as cross-boundary optimizations. Furthermore, applying DONT_TOUCH to the target IP has little effect on the optimization of the user's surrounding circuitry. The user's surrounding circuit will still be fully optimized, and in fact, can still leverage knowledge of the content of the trusted IP.

Once the user has instantiated the trusted IP netlist in their design they can perform implementation (place & route) as usual. After implementation, the user uses the write_verilog command to extract a Verilog netlist from the implemented design. Like the trusted IP netlist, this user design netlist contains Xilinx primitives. As in the Trusted IP Flow, we assume that Vivado faithfully retrieves the correct netlist, while acknowledging that a compromised version of Vivado could potentially retrieve a spoofed netlist (addressed in Section 5).

4.1.3 Functional Comparison The final phase of our implementation is the comparison flow, where Cadence Conformal, a formal equivalence checker[1], is used to compare the trusted IP netlist and the user's netlist.

4.2 Challenges and Limitations of Functional Assurance

One benefit of providing assurance at the functional level is that many of the challenges present in the physical assurance process (discussed previously in Section 3.2), such as global resources or Pblock location, no longer pose an issue. However, we did encounter a couple of new challenges:

Renaming Rules: When exporting the instantiated IP as Verilog netlists, Vivado would sometimes unexpectedly rename input and output ports. This happened to only a handful of ports on only a few of the 53 instantiated IP blocks. This unexpected renaming caused problems downstream in the assurance process—because Conformal relies on the primary input and output names of the trusted and instantiated IPs to match, such renaming breaks

[1]For the purposes of this article, it is assumed that logical equivalence implies functional equivalence. Although logically-equivalent netlists may differ in treatment of certain don't-care conditions, for example, it is assumed that these differences, if they occur, do not affect the functional behavior of the IP in any meaningful way.

Conformal's ability to determine equivalence. We overcame this challenge by writing a netlist parser that would find where instances of renaming had occurred. The parser would then generate a list of TCL commands which were issued to Conformal to inform it of equivalence between renamed ports.

Unused Outputs: One special case that we encountered was when certain outputs of the IP were unused in the parent design. Normally in such a case Vivado would simply remove the unused internal pins and associated logic; however, because we applied a `DONT_TOUCH` attribute to all pins, the unused internal pin remained in the design, and Vivado drove this pin by a constant 0. However, when processed by Conformal, this internal pin was optimized away, not driven by constant 0. This subtle and harmless discrepancy was enough to cause equivalence checking to fail. To handle this we included a script in our flow that locates the unused IP outputs and auto-generates a list of TCL commands that would instruct Conformal to ignore the unused outputs when making the comparison.

4.3 Experiments with Functional Assurance

Similar to the experiments discussed in Section 3.3, we wanted to ensure that the proposed technique could reliably determine equivalence for an unmodified IP (ie, no false positives) and reliably detect non-equivalence for a modified IP (ie, no false negatives).

For our experiments we used the same set of designs as in the previous section (LEON3 and a collection of synthetic benchmarks, Table 1), and used the same approach of choosing one IP at a time to serve as the "trusted IP".

4.3.1 Equivalence of Unmodified Circuits After improving our compilation flow to address the issues discussed in Section 4.2, every benchmark/trusted IP in our suite could be run through our functional assurance flow (Section 4.1/Figure 5) and result in a perfect match between the trusted IP and the instantiated IP.

4.3.2 Detecting Circuit Modifications Similar to the physical assurance testing, we developed a sensitivity analysis to demonstrate that our functional assurance flow successfully detects unwanted modifications. We tampered with the netlist of the designs in the following ways:

- Pick a random Lookup Table (LUT) in the instantiated IP and modify its logic function.
- Leak a random wire in the instantiated IP to a secretly added backdoor port.

We tampered with each of our instantiated IPs once for each of the above modifications. This gave us a total of 106 tampered designs which we used to test the sensitivity of our approach. Our approach caught all of the malicious modifications that we made.

4.4 Impact of Functional Assurance on QoR

As with the physical assurance flow, for each "trusted IP" and parent benchmark listed in Table 1, we performed compilation using our proposed flow, and compared slice utilization against a standard Vivado compilation. The results are shown in Figure 6.

In most cases, the resource utilization increases, which is expected due to the fact that the flow prevents any cross-boundary optimization between the trusted IP and the rest of the design. The `DONT_TOUCH` directives also likely prevent any physical optimizations that may have normally taken place post-synthesis. On average, designs that used the functional assurance method used 2.8% more slices than the baseline design. The greatest performance decrease was observed with the `sudoku` design, which was 27.7% larger than its baseline design. On the other end of the spectrum was the `msp430_vhdl` design, which was actually 7.6% smaller than its baseline design. Again, this unexpected reduction can occur in some

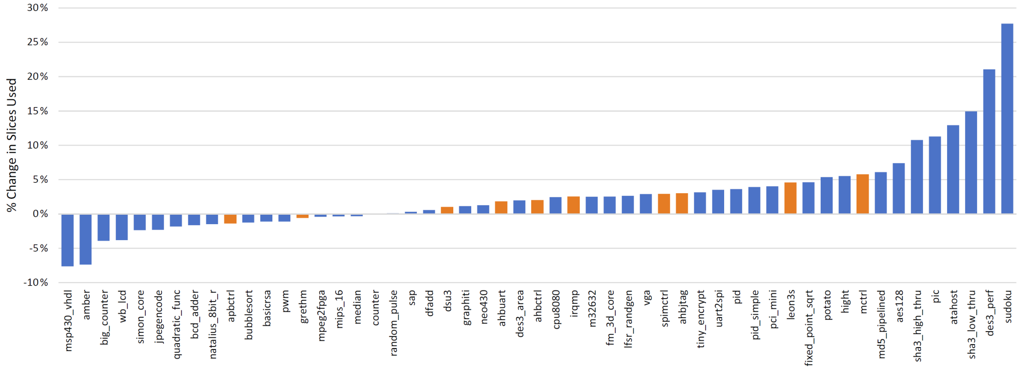


Fig. 6. Impact of functional assurance flow on slice utilization.

cases due to the stochastic nature of the CAD tools. It makes sense the penalty varies from design to design as many optimizations, such as logic minimization based on constant or unconnected inputs, will depend on the surrounding design.

Of the 10 experiments run on the LEON3 benchmark, all 10 passed timing in both the baseline and Pblock-based designs. Of the 43 experiments created from the synthetic benchmarks, no designs failed our synthetic 10ns timing constraint after its baseline design had passed timing. This suggests that our approach does not readily cause a design to fail timing if its baseline comparison design passed timing, and that the effect of the functional assurance approach on timing is likely small.

5 Bitstream-Level Design Assurance

The physical and functional IP assurance techniques presented in Sections 3 and 4 relied upon the CAD tool to report details of the implemented design. If the CAD tool were compromised, or if the generated bitstream were modified post-generation, these approaches would not be effective. This limitation has motivated our most recent work, described in this section, which is to explore techniques to verify equivalence at the bitstream level.

This work aims to address this limitation by leveraging open-source bitstream tools which can convert a binary bitstream back to a human readable netlist. This netlist is then compared against the original design.

The reverse-netlist tool we employ in this section of our work is part of *Project Icestorm* [7], which only supports Lattice iCE40 FPGAs. At the time of this work, and to our best knowledge, this was the only FPGA bitstream to netlist tool available[2]. As such, the tool flows discussed in this section target the much smaller Lattice iCE40 family of FPGAs, rather than the Xilinx 7-series FPGAs targeted in the earlier sections.

5.1 New Challenges Imposed by Bitstream-Level Assurance

Trying to perform comparison with a bitstream-reversed netlist introduces several new challenges not present in our earlier work. The paramount challenge is that the reverse-engineered netlist has no net names from the original design, nor does it have any semblance

[2]At the time of publishing there is now another tool that supports Xilinx FPGAs, the Symbiflow fasm2bels tool, https://github.com/SymbiFlow/symbiflow-xc-fasm2bels.

of logical partitioning or separation between the different IP or submodules that made up the original design.

The lack of net names makes the equivalence checking challenging, as the formal verification tools have no starting point on which to perform comparison. Fortunately the placement constraints file can be leveraged to extract net names of the primary input and output pins. In fact, bitstream reversal tools typically allow you to provide the file of original placement constraints, and the produced netlist will automatically have primary input and output signal names restored. However, all internal net names are inevitably lost.

Not being able to extract out IP or submodules from the reversed netlist means that equivalence checking must now be performed on the entire design. This introduces a fundamentally different problem that what we tackled in earlier sections. While our earlier work only focused on vetting trusted IP that were inserted into a larger design, we are now forced to perform equivalence checking on the entire design, including the user's own logic.

Not only does this mean we are forced to tackle larger circuit sizes, it also defeats some of the assumptions we made in our earlier work. For example, in our functional IP assurance flow described in Section 4, our solution assumed that a trusted third party would presynthesize a design, and then vet that the optimized, technology mapped netlist was safe. This assumption can no longer be the case if we instead are trying to determine equivalence for the entire design, which now contains user-created content.

In our initial testing we explored a straightforward approach passing several of our benchmark designs through the commercial Lattice FPGA compilation flow, iCECube2, to produce a bitstream, and then using *Project Icestorm* to convert the bitstream back into a Verilog netlist. The original RTL and reversed netlist are then compared using *Cadence Conformal*. This flow is illustrated in Figure 8a. Unfortunately, this basic approach did not work well, and very few of our RTL benchmarks were reported as equivalent to the produced bitstream. This is not surprising given the significant transformations that take place during the CAD flow.

5.2 BYU FPGA Assurance Tools (`bfasst`) Framework

Given the fact that the basic commercial CAD flow broke our equivalence checking, we set out to explore alternative CAD flows that would prove more effective. However, composing custom CAD flows and running them for large sets of benchmark designs is very time consuming. In earlier sections we presented two different custom CAD flows for IP assurance. These CAD flows required significant manual effort to execute the custom CAD stages for large set of benchmarks. As such, it was our goal to develop a more flexible framework, that would allow us to programmatically compose custom CAD flows and collect results on a large number of benchmark automatically.

The framework we developed is implemented as a Python package, which we refer to as the BYU FPGA Assurance Tools (`bfasst`) package. Figure 7 provides an overview of this framework, and illustrates how different modules can be plugged in to perform the various stages of compilation. Our custom CAD flows all follow the same pattern: the user's RTL design is synthesized, optimized, and implemented to generate a bitstream. This bitstream is then processed with an open-source tool, such as Project Icestorm to generate a netlist representation of the bitstream. We then use formal verification to validate the netlist against previous stages in the synthesis process. Doing so allows us to identify if the bitstream is an accurate representation of the design at these stages.

While the primary motivation behind this framework is for design assurance purposes, it is likely useful by others in the FPGA community who are interested in composing and

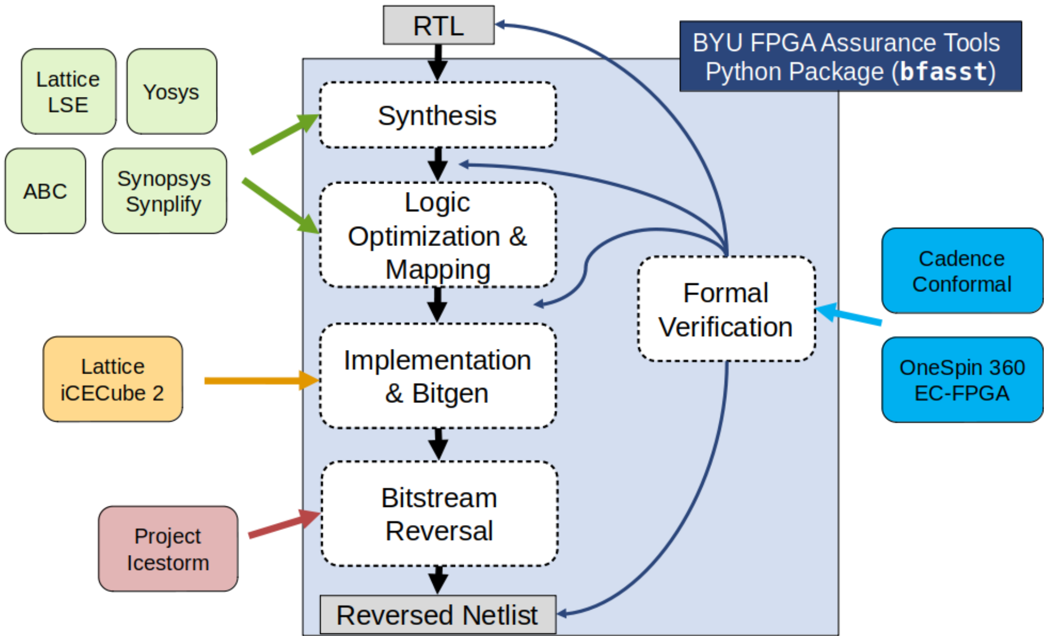


Fig. 7. bfasst offers a modular approach to synthesis, bitstream reversal, and verification, allowing different tools to be swapped in for the various validation steps.

evaluating different CAD flows. The tool is open source, and available at github.com/byuccl/bfasst.

BFASST is implemented as a Python package, the heart of which is a collection of classes that are each responsible for executing one tool for a single stage of the FPGA CAD flow. These classes are subclassed from base classes for each stage of the CAD flow, meaning that, for example, all synthesis-type tools adhere to a common interface. This approach means that these tools can then be stitched together to form custom CAD flows with relative ease. Stitching these tools together is done by implementing a custom Python function that instances and connects the various chosen tools. Listing 2 provides the Python code for creating the flow shown in Figure 8a.

This custom flow function is then registered with BFASST, which provides command-line scripts to invoke the CAD flow. Users can call a simple script (`run_design.py <flow_name> <design_path>`), that executes a single benchmark circuit and CAD flow, or alternatively, users can build experiment configuration files that specify a list of benchmarks and CAD flows to execute (`run_experiment.py <config.yml>`), which will run all flow/benchmark combinations and generate a report to aggregate the results.

It is worth noting that while arbitrary CAD flows can be created, the selected tools must still be compatible with each other. For example, the implementation and bitgen tool must receive a netlist in compatible format from the logic optimization and mapping tool.

One challenge we encountered is that we wanted to use the same benchmark designs on multiple different commercial and open-source tools (Lattice, Xilinx, Yosys, etc); however, each of these uses proprietary project management files to list design sources, properties, libraries, etc. As such, we also developed a simple configuration file (YAML format) that we

Listing 2. BFASST Flow Example

```
def flow_ic2_lse_conformal(design, build_dir):
    # Run Icecube2 LSE synthesis
    synth_tool = IC2_LSE_SynthesisTool(build_dir)
    status = synth_tool.create_netlist(design)
    if status.error:
        return status

    # Run Icecube2 implementation
    impl_tool = IC2_ImplementationTool(build_dir)
    status = impl_tool.implement_bitstream(design)
    if status.error:
        return status

    # Run Project Icestorm bitstream reversal
    reverse_bit_tool = Icestorm_ReverseBitTool(build_dir)
    status = reverse_bit_tool.reverse_bitstream(design)
    if status.error:
        return status

    # Run Cadence Conformal
    compare_tool = Conformal_CompareTool(build_dir)
    status = compare_tool.compare_netlists(design)

    return status
```

use in our framework to describe a design's sources and properties. These design configurations files are parsed by our framework, populated into a `Design` class, and provided to the various tool wrappers. This means that benchmark designs only need to be setup once in our framework, and then can be processed by all the supported tool flows.

5.3 Choosing an Effective CAD Flow for Equivalence Checking

We initially leveraged our BFASST framework to test a standard commercial flow, as shown in Figure 8a. As mentioned previously, this equivalence checking failed for the vast majority of our benchmarks; only 12% of produced bitstreams were reported to be equivalent to their original RTL.

The primary issue here is that we are attempting to compare a netlist representation of the design against the RTL. When we use Cadence Conformal for comparison, Conformal tries to match key points (registers, I/O, etc.) between the two designs, and then compares the logic between key points. However, the synthesis tools will make optimizations to the design during synthesis and implementation that can change the logic between key points without changing the functionality of the design (e.g. register retiming or removing redundant registers). Even without substantial optimizations, it is entirely possible that the design representation in the RTL is different enough from the design representation in the netlist that they cannot easily be compared.

We leveraged our BFASST framework to explore various CAD flows, in hopes of locating a CAD flow that could effectively be used for verification. After testing several different

CAD flows, we settled on the flow shown in Figure 8b, which consisted of the following modifications:

(1) **Rather than comparing the reverse netlist against the original RTL, we compare it against a netlist that has already undergone synthesis, logic optimization and technology mapping.** While it would be possible to export such a netlist out of the commercial flow, a major motivation of this work is to establish an equivalence checking flow that does not need to trust the closed-source commercial tools. As such, our general approach is to use a open-source synthesis tool that can pre-synthesize and optimize the design prior to entering the commercial CAD flow.

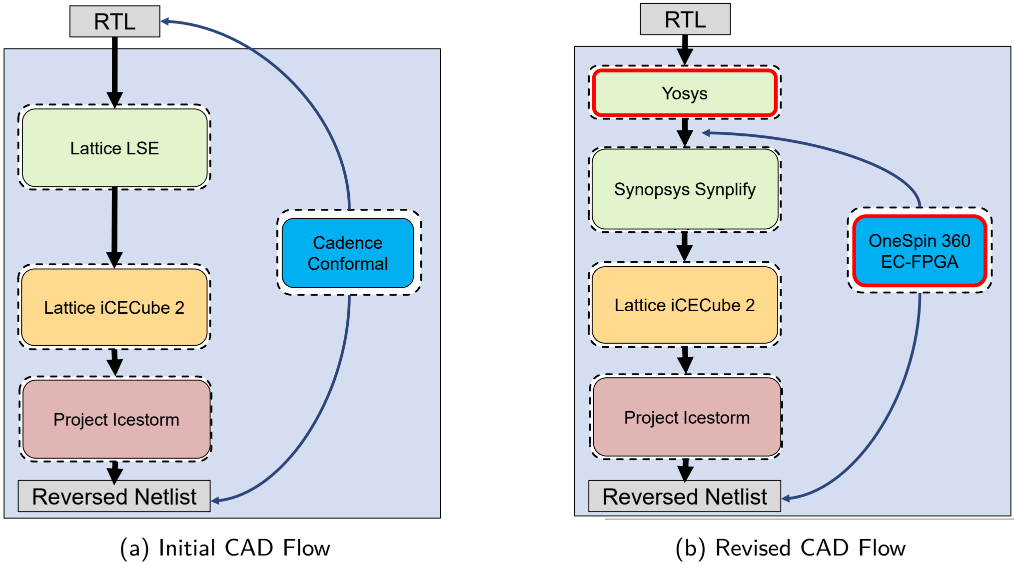
In our Lattice-based tool flow, we chose to use Yosys [28], an open-source synthesizer capable of compiling Verilog HDL to a technology mapped netlist targeting Lattice iCE40 FPGAs. While we recognize that using an open-source tool does not automatically mean it is free from malicious behavior, it is a step in the right direction, and should an organization desire, they could invest time into inspecting the source code and establishing some level of trust, or alternatively, capable organizations could develop their own synthesis or bitstream reversal tools if they did not trust existing open-source tools.

(2) **We configured the Lattice toolchain to use the Synopsys Synplify synthesis tool** (it offers both Synplify and LSE synthesis tools). Even though the netlist provided by Yosys is already synthesized and mapped to primitives, you must still use a synthesis front-end with the iCECube2 toolchain. When using the Synplify front-end, more of our benchmark designs passed the verification flow. While further investigation is required, it seems that this tool was less likely to modify our already technology mapped netlist.

(3) **To further enhance our equivalence checking flow, we elected to use a different comparison tool, OneSpin 360 EC-FPGA** [8]. Unlike Conformal, this tool is specifically designed to target FPGA design flows, and is advertised as capable of determining equivalence in the presence of additional optimizations, such as FSM re-encoding, pipelining, retiming and others.

Using this new flow, we were able to establish equivalence for 100% of our benchmark designs (ie. eliminate all false positives of design mismatches). While we believe this is a terrific result, and a key step forward toward building general-purpose bitstream-level equivalence checking tools, a few key limitations should be noted:

1. Our flow is currently dependent upon having a tool to convert the bitstream back into a netlist. This means that our work is limited to Lattice iCE40 FPGAs, a small fraction of the entire FPGA market. However, many ongoing projects such as Project X-Ray [23] and Project U-Ray [24] are working to establish such tools for larger families of FPGAs, and we are currently working to expand our flow to include these tools.
2. Our current flow uses the Yosys synthesis tool. Yosys only accepts Verilog HDL as input, as such, the number of our benchmark designs we could pass through this flow is much smaller than what we used in earlier sections (Table 1). The Verilog benchmarks we used the bitstream-level assurance experiments in this section of the article are found in Table 2. The *Orig* column in the table provides benchmark statistics for the baseline commercial flow (flow from Figure 8a).
3. The iCE40 FPGA family consists of very small FPGA parts. As can be seen in Table 2, our benchmark designs are limited in size to only thousands of LUTs, orders of magnitude smaller than what is available in the largest modern FPGAs. Even though the designs are small, the comparison runtime is still significant (as discussed later in



(a) Initial CAD Flow

(b) Revised CAD Flow

Fig. 8. Two CAD flows created using the BFASST framework.

this section). As such, we recognize that there are still significant scalability issues to tackle in future work.

(4) Our proposed flow relies heavily upon open source tools, particularly to first perform initial synthesis and logic optimization to produce a netlist used for later comparison, and second, to produce a reversed netlist from the final bitstream. Many organizations may be hesitant to trust and integrate these unverified open-source tools into their toolchain. In such cases the organization may have to produce these tools in-house (for example, we know of a few research groups that have created their own bitstream to netlist tools internally), or advocate for this functionality to be provided by existing commercial parties.

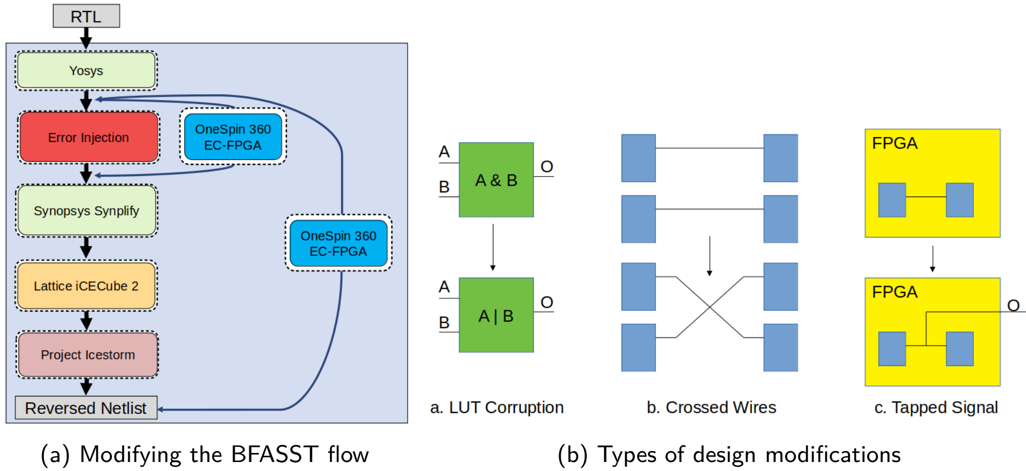
5.4 Detecting Circuit Modifications

To validate this flow, we used a similar approach to earlier sections (Sections 3.3.3 and 4.3.2), and modified the design in some small but malicious way.

To do this we modified our baseline CAD flow (Figure 8b), and added an additional step that injects some modification into the design (Figure 9a). The error injection phase is inserted immediately after the Yosys synthesis stage, allowing us to make modifications to the design in the Yosys netlist. By making design modifications to the Yosys netlist, we can simulate various types of design modifications that may be performed by an attacker.

In our experiments we explore three types of design modifications (Figure 9b):

(1) *LUT Corruption.* We corrupt a LUT in the design by flipping a bit in the LUT init string in the netlist, changing the logic function of that LUT. We perform two variants of this error injection – one where we modify a single LUT bit in the design, and one where we select and flip five bits in the design. This second test is done to increase the likelihood of meaningfully impacting the design with this error type.



(a) Modifying the BFASST flow

(b) Types of design modifications

Fig. 9. Injecting design modifications

(2) *Crossed Wires.* We take two random signals and swap their sources, so that their fanout is fed by the wrong source.
(3) *Signal Tapping.* We pick a random non-I/O signal in the design, and tap it by adding an extra output port to the design that is driven by the selected signal.

As can be seen in Figure 9a, we perform two equivalence checks with OneSpin 360 for every injected error. First, we perform a comparison between the netlist generated by Yosys and the corrupted netlist generated by the error injection tool. We then perform a second comparison between the Yosys-generated netlist and the reverse-engineered netlist (created from the final bitstream).

The first comparison allows us to verify that the design modification meaningfully impacts the design. Because the error injection tool injects its errors at random, it is possible for it to modify the design in a way that does not actually affect the functionality of the design. For example, if the sources of two wires are swapped, but the two wires are both part of the same AND reduction operation, then the swap has no impact on design functionality. A second, more common example, is that flipping a bit in a LUT's init string may not always affect the functionality of the design, as not all LUT inputs are always used. The second functional comparison – between the Yosys netlist and the reverse-engineered bitstream netlist – is the comparison the BFASST tool would perform under normal operation.

Between the two comparisons, we can determine if a design modification actually impacts the design, and if it does, whether or not we can detect the error once the design has been fully synthesized and reverse-engineered.

For each benchmark in Table 2 we completed 20 instances of the design modification flow (five each of single-bit LUT corruption, five-bit LUT corruption, crossed wires and tapped signal). In our testing, we were always able to detect circuit modifications that changed the functionality of the design.

5.5 Impact of Bitstream-level Assurance on Runtime and QoR

5.5.1 Runtime The first overhead we investigated was the runtime of performing the comparison using the commercial equivalence checking tool, across our suite of benchmarks.

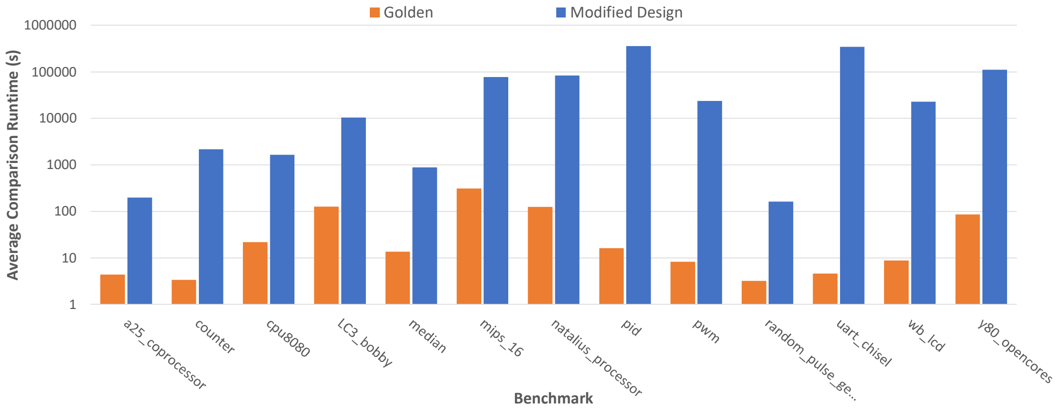


Fig. 10. Comparison runtimes for the golden unmodified design, and the modified design where a malicious design modification is made. For the *Modified Design* series, the value represents the average runtime across all design modification runs.

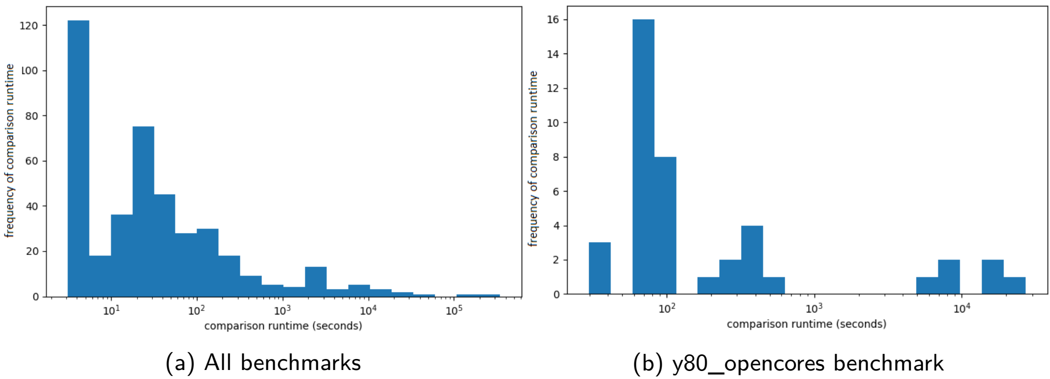


Fig. 11. Histogram showing distribution of runtimes. This includes the golden comparison, plus both comparisons (shown in Figure 9a) for each of the 20 design modification runs.

Figure 10 provides these results. The *Golden* series, colored in orange, provides the comparison runtime for the various benchmarks without any design modifications. Runtimes vary from 3.2s to 308s, indicating that for even these small designs, comparison runtime may be significant.

When designs are modified, the comparison runtimes become much larger. This is possibly because the tool must now exhaustively search across all possible mappings of internal state to conclude that there is no mapping that produces an equivalent result. The runtimes of the equivalence checking, averaged across all design modification runs, is shown in the *Modified Design* series in Figure 10. These averages range from 8.0s to 18039s (about 5 hours) (it should be noted that a few designs are excluded from this figure as they had instances where the equivalence checking of the modified design never returned, even after more than a week of execution time). Runtimes vary drastically from one design modification to the next; Figure 11a provides a histogram of runtimes for all benchmarks. Figure 11b filters this

Table 2. Area overhead of proposed bitstream-level assurance flow

Benchmark	**# LUTs**		**# FFs**		**# CARRY**		**# BRAM**	
	Orig	*New*	*Orig*	*New*	*Orig*	*New*	*Orig*	*New*
a25_coprocessor	219	254	171	171	0	0	0	0
counter	34	65	32	33	32	30	0	0
pid	1505	1308	394	396	0	0	0	0
pwm	372	570	145	146	75	150	0	0
random_pulse_gen	40	38	33	33	0	0	0	0
simon_core*	213	353	167	281	21	38	2	0
uart2spi	916	1143	429	456	32	70	0	0
wb_lcd	206	242	85	97	19	18	0	0
LC3_bobby	568	687	187	187	45	44	1	1
natalius_processor	423	532	111	124	25	49	9	9
mips_16_fixed	863	931	298	300	23	43	1	1
uart_chisel	129	167	60	64	24	30	0	0
median_fixed	1052	1076	45	124	121	483	0	0
cpu8080*	2466	2349	243	243	331	337	1	0
pci_mini	565	715	340	369	5	8	0	0
y80_opencores	4018	2972	407	407	86	90	0	0
aes_opencores	3149	3202	924	922	31	34	0	0
Mean	476.2	554.8 +16.5%	160.9	181.4 +12.7%	51.2	83.8 +63.7%	0.8	0.6 -25%
Mean (excluding *)	450.2	519.3 +15.3%	156.2	172.7 +10.6%	34.5	69.9 +102.6%	0.7	0.7 0%

Note: *Orig* indicates the original commercial flow (Figure 8a), while *New* indicates the proposed flow that enables equivalence checking (Figure 8b). Means are provided using geometric mean for LUTs/FFs, and arithmetic for CARRY/BRAM (which contain zeros). The final row excludes the two benchmarks where the flows differ in number of BRAMs used, as it appears in these cases the new CAD flow is implementing some BRAM content with other resources.

data to only the *y80_opencores* benchmark, and demonstrates that even within the same benchmark, runtimes vary by over 100x depending on how the design is modified.

While the modified design runtimes are lengthy (and in a few cases never complete), this only exists in the rare case when the design has been modified, and the comparison tools struggle to find an equivalence. In a normal design flow, the design would be unmodified, and a designer could gain assurance of equivalence in a much shorter runtime. It would typically only be in the case of malicious design modification that these much longer runtimes would manifest.

5.5.2 Area Overhead In most cases, pre-synthesizing the design to a technology mapped netlist before entering the commercial flow comes at a cost. Table 2 provides the per-benchmark resource usage, post-implementation, for the original commercial flow (Figure 8a) and our proposed flow (Figure 8b). The LUT and FF usage values are also visualized in Figure 12. On average, our proposed flow increases LUT usage by 15–17% and FF usage by 11–13%.

Table 3. Comparison of Assurance Approaches

Goal	Area Overhead	Advantages	Limitations/Disadvantages
Physical IP Assurance (Section 3)			
Verify placement and routing of trusted IP in implemented parent design.	+4.7% Slices	— Detects malicious modifications to design during CAD flow. — Detects any discrepancy, including physical-level attacks. — Simple and fast comparison process.	— Requires IP provider to provide placed and routed Pblock. — Provided IP restricted to implemented location and target part. — Restrictions on global resource (e.g. BUFG cannot be placed within Pblock). — Does not detect modifications to the bitstream post-generation. — Trusts CAD tool to output true implementation details.
Functional IP Assurance (Section 4)			
Verify functional equivalence of trusted IP in implemented parent design.	+2.8% Slices	— Detects malicious modifications to design during CAD flow. — IP provider creates optimized netlist; more flexible than providing a placed and routed Pblock.	— Comparison process requires commercial formal verification tool. — Does not detect modifications to the bitstream post-generation. — Trusts CAD tool to output true implementation details.
Bitstream-Level Assurance (Section 5)			
Verify equivalence of bitstream to original RTL design, leveraging pre-synthesis with open-source tool.	+15% LUTs	— Detects malicious modifications to design during CAD flow, or to bitstream post-generation. — Operates on entire design, not just a single IP. — Does not rely on trusting commercial CAD flow.	— Comparison process requires commercial equivalence checking tool. — Requires a bitstream to netlist tool, which have only been developed for certain FPGA vendors and families. — Trusts open-source CAD tool to output true implementation details. — Longer comparison runtimes; performing equivalence checking on entire design may not be scalable to larger design sizes.

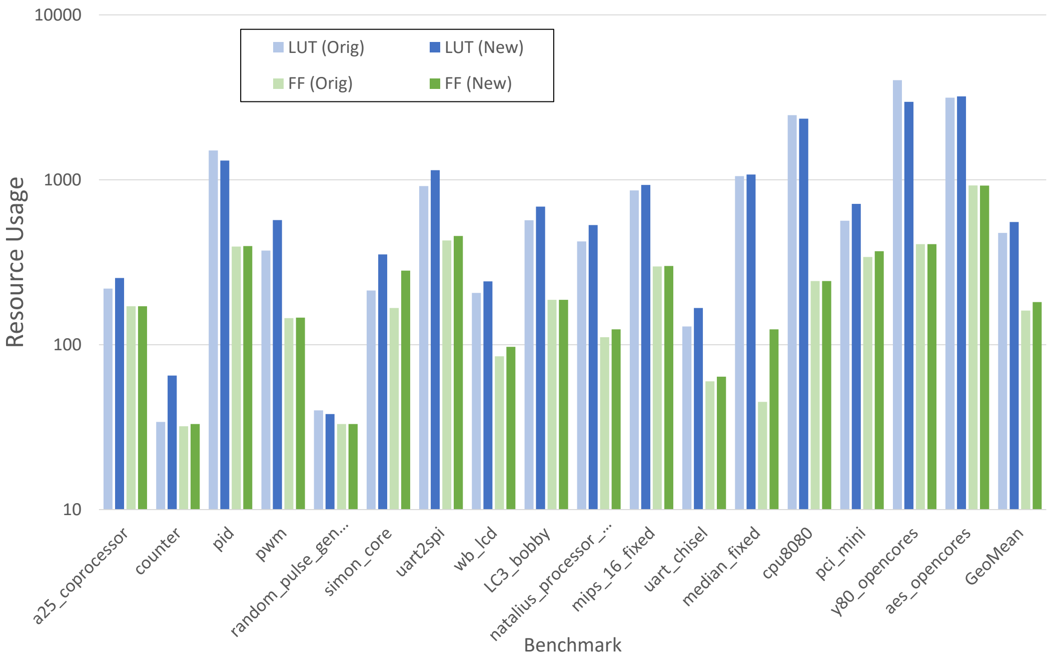


Fig. 12. Resource Usage

6 Comparing Different Assurance Flows

Table 3 provides a summarized comparison of the different assurance flows. The *Physical IP Assurance* flow provides the greatest guarantee that the trusted IP is implemented correctly in the design, as it guarantees that every placement and routing matches the provided implemented IP. This prevents attacks that could bypass a functional equivalence check, such as creating very long routing paths that violate timing constraints and cause the IP to not function correctly. However, this guarantee comes at a price, as the IP provider must supply the IP already implemented. This requires more burden on the IP provider to vet that the implemented design is correct and safe. It also hinders use by the designer, as the IP would be locked to a certain location and FPGA part. In all likelihood, those wanting to pursue this option would require that the IP provider implement the IP for several chip locations and/or parts. This flow also has additional restrictions due to what global resources can legally be placed within a *Pblock*; for example, the IP could not contain a *BUFG* or other similar resources. In addition, the assurance flow comes at a cost. Our experiments indicated that slice usage increases by almost 5% on average.

The *Functional IP Assurance* flow addresses some of these issues by allowing the IP provider to create a pre-optimized netlist. Compared to Physical IP Assurance, this provides greater ease and flexibility to the user as they do not not have to physically partition their design, but can rather just instantiate the trusted IP netlist. In addition to the ease of use improvements, it also incurs less overhead; our experiments found just a 3% average increase to number of slices. However, it should be noted that this additional flexibility comes at a price; it only protects against tampering at the logical (*i.e.* RTL, gatelist, or netlist) level of the IP. It does not protect against tampering at the implementation level of the design, *i.e.* placement and routing. For example, a malicious CAD tool or attacker could tamper with

the IP by placing its cells unnecessarily far apart from each other, thus slowing down timing and degrading the IP's performance.

Both of these IP-centric assurance flows take the approach of having the CAD tool report the IP as part of the implemented final design. This introduces some trust in the CAD flow, and also means that modifications to the bitstream post-generation would be undetected. Our final approach, *Bitstream-Level Assurance* addressed this limitation by leveraging open-source bitstream to netlist tools to generate a netlist for comparison against the original design.

However, the reversed netlist no longer contains any internal net names, hierarchy, or module partitions, meaning that the comparison processes must be performed on the entire design, and the runtimes are significant for even small designs.

The overhead we measured for the bitstream-level assurance (15% increase to LUTs) is larger than the other approaches; however, its important to recognize that these earlier approaches only protected a single IP in the overall design, while the bitstream-level assurance targets the entire design. This would likely account for the larger overhead observed.

7 Conclusion and Future Work

In conclusion, our different assurance flows demonstrate that with some modifications to the default FPGA CAD flow, it is possible to assure designers that their compiled hardware design is equivalent to their trusted source design and/or IP. All of the processes we described in the article are fully automated, suggesting that designers and engineers can easily determine the integrity of their designs without needing hardware security expertise.

The different approaches discussed in the paper have various strengths and weaknesses, and organizations may choose different approaches based on their security goals and needs. Overall, the different approaches all come at an area cost (3–16% increase in logic); however, we believe that in most cases this cost would be acceptable for the assurances that can be provided.

Although the results we have obtained are promising, this is still only a step toward the larger goal of assuring the FPGA design process for arbitrary designs and FPGA families. A number of key challenges remain:

(1) The runtimes we encountered were significant, and novel approaches will be needed to scale this work to larger designs. There is active work in developing better graph isomorphism (graph matching) algorithms [29], [30] that will hopefully enable more scalable equivalence checking tools in the future.
(2) For certain benchmarks, the area overheads can be substantial, exceeding 20% overhead. In some cases a designer may not have enough spare logic on their device to use these techniques. It would be interesting to investigate what assurance techniques could be applied in such resource-limited scenarios.
(3) The functional and bitstream-level approaches we used required performing equivalence checking against an optimized and mapped netlist, rather than the original RTL. While capable organizations could ensure that this optimized netlist remains free from malicious content, it requires time and effort. In future work it would be beneficial to explore whether verification could be successful if attempted in smaller increments. For example, comparison could be done before and after each individual stage of the flow, rather than our proposed techniques which performed equivalence across many steps of the CAD flow.

(4) Bitstream documentation is limited, and further open-source tools will be required to provide these assurances for a wider set of FPGA families.
(5) As hardware design shifts to higher abstraction levels (high-level synthesis tools, domain-specific language compilers) the assurance problem grows as it will become more challenging to prove equivalency with these higher-level input descriptions.

While several challenges remain in the face of general FPGA design assurance, we believe the outlook is still positive. The work presented in this paper would not have been possible without the advancements in open source tools over the past few years, and we believe growing traction in the open source community will likely open the door to further verification efforts in the future.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] C. Wolf. (). Project IceStorm, Project IceStorm, [Online]. Available: http://www.clifford.at/icestorm/ (visited on 06/23/2020).

[8] OneSpin. (). 360 EC-FPGA – OneSpin solutions. Library Catalog: www.onespin.com, [Online]. Available: /products/360-ec-fpga (visited on 06/25/2020).

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. Wolff1, C. Papachristou, K. Roy, and S. Bhunia, “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.

[16] 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.

[17] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series FPGAs," in *USENIX Conference on Security Symposium*, 102, USA: USENIX Association, Aug. 12, 2020, pp. 1803–1819.
[18] H. Yu, H. Lee, S. Lee, Y. Kim, and H.-M. Lee, "Recent advances in FPGA reverse engineering," *Electronics*, vol. 7, no. 10, p. 246, Oct. 2018.
[19] K. Matas, T. M. La, K. D. Pham, and D. Koch, "Power-hammering through glitch amplification – attacks and mitigation," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2020, pp. 65–69.
[20] B. L. Hutchings, J. Monson, D. Savory, and J. Keeley, "A power side-channel-based digital to analog converter for xilinx FPGAs," in *Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 26, 2014, pp. 113–116.
[21] I. Hadžić, S. Udani, and J. M. Smith, "FPGA viruses," in *Conference on Field Programmable Logic and Applications (FPL)*, P. Lysaght, J. Irvine, and R. Hartenstein, Eds., 1999, pp. 291–300.
[22] T. Gaskin, H. Cook, W. Stirk, R. Lucas, J. Goeders, and B. Hutchings, "Using novel configuration techniques for accelerated FPGA aging," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, ISSN: 1946-1488, Aug. 2020, pp. 169–175.
[23] https://github.com/SymbiFlow/prjxray, *SymbiFlow/prjxray*, Jun. 23, 2020.
[24] https://github.com/SymbiFlow/prjuray, *SymbiFlow/prjuray*, Jul. 21, 2020.
[25] D. R. E. Gnad, S. Rapp, J. Krautter, and M. B. Tahoori, "Checking for electrical level security threats in bitstreams for multi-tenant FPGAs," in *International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 286–289.
[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.
[27] K. Kępa, F. Morgan, K. Kościuszkiewicz, L. Braun, M. Hübner, and J. Becker, "Design assurance strategy and toolset for partially reconfigurable FPGA systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 1, 4:1–4:26, Dec. 1, 2010.
[28] C. Wolf, *Yosys open SYnthesis suite*.
[29] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, Jan. 1, 2014.
[30] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, Oct. 2004.