Toward FPGA Intellectual Property (IP) Encryption from Netlist to Bitstream

DANIEL HUTCHINGS, Brigham Young University, USA
ADAM TAYLOR, Brigham Young University, USA
JEFFREY GOEDERS, Brigham Young University, USA

Current IP encryption methods offered by FPGA vendors use an approach where the IP is decrypted during the CAD flow, and remains unencrypted in the bitstream. Given the ease of accessing modern bitstream-to-netlist tools, encrypted IP is vulnerable to inspection and theft from the IP user. While the entire bitstream can be encrypted, this is done by the user, and is not a mechanism to protect confidentiality of 3rd party IP.

In this work we present a design methodology, along with a proof-of-concept tool, that demonstrates how IP can remain partially encrypted through the CAD flow and into the bitstream. We show how this approach can support multiple encryption keys from different vendors, and can be deployed using existing CAD tools and FPGA families. Our results document the benefits and costs of using such an approach to provide much greater protection for 3rd party IP.

ACM Reference Format:
Daniel Hutchings, Adam Taylor, and Jeffrey Goeders. 2024. Toward FPGA Intellectual Property (IP) Encryption from Netlist to Bitstream. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2024), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

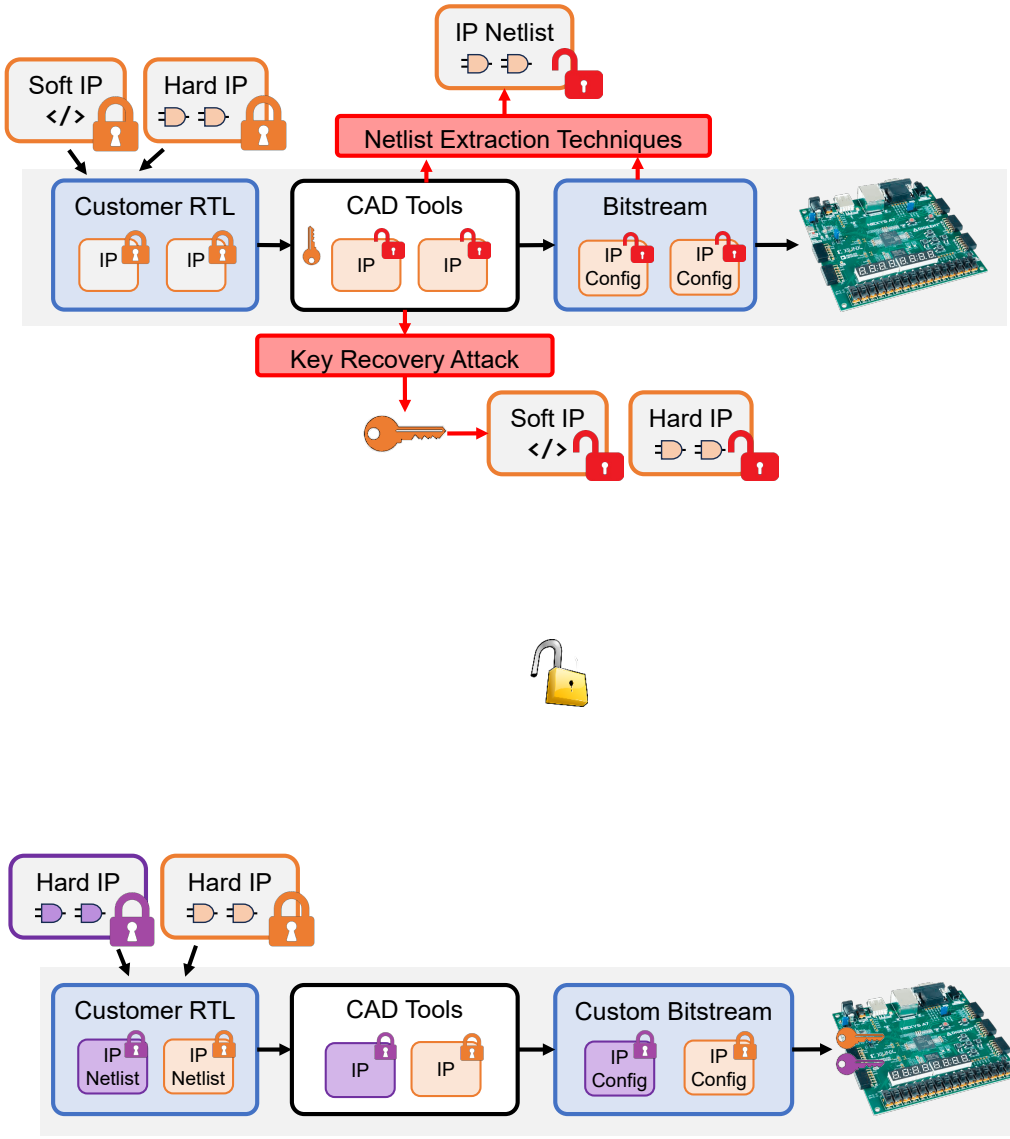
1 INTRODUCTION

Modern field-programmable gate array (FPGA) design flow often consists of creating large complex systems of many interconnected intellectual property (IP) blocks. These IPs could be obtained from the FPGA vendor, from other departments within the designer's organization, or licensed from third parties. Whether for reasons of national security, maintaining trade secrets, or mitigating risk of theft and unlicensed reuse, the IP designer may want to keep the IP confidential when distributing it for use in a customer's design.

While current commercial FPGA design tools follow an industry-standard method to encrypt IP details (IEEE-1735-2014 [1]), this approach is not robust enough to prevent an attacker from gaining access to the full details of the IP. For example, in [2], Speith et al. demonstrate weaknesses in the IEEE-1735 approach that allow them to obtain the IP encryption keys for all major electronic design automation (EDA) vendors, including the most popular FPGA vendors. Once these keys are obtained, the authors are able to, "decrypt, modify, and re-encrypt all allegedly protected IP cores designed for the respective tools, thus leading to an industry-wide break." Such an attack allows the IP user to gain access to the full details of the IP, including the ability to reconfigure the IP, modify it, or even sell it as their own.

Authors' addresses: Daniel Hutchings, danh444@byu.edu, Brigham Young University, 450EB, Provo, Utah, USA, 84602; Adam Taylor, adftaylo@byu.edu, Brigham Young University, 450EB, Provo, Utah, USA, 84602; Jeffrey Goeders, jgoeders@byu.edu, Brigham Young University, 450EB, Provo, Utah, USA, 84602.

1936-7406/2024/1-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX



(b) Proposed IP encryption approach. IP remains partially encrypted through the CAD flow and into the bitstream. The IP is protected from inspection and theft from the IP customer. Multiple IP vendor keys can be supported, although only hard (post-synthesis netlist) IP are supported.

Fig. 1. IP Encryption Approaches. In the diagrams above, both soft IP and hard IP are shown. Soft IP (illustrated with the </> symbol) refers to RTL code that may be parameterizable to generate different Hard IP netlists. Hard IP (illustrated with AND-gate symbols), refers to a post-synthesis structural netlist that contains a collection of connected FPGA primitives.

Even if an IP customer does not have the capability for a sophisticated attack that recovers encryption keys from program memory, current FPGA tools expose significant netlist information of the encrypted IP, allowing a user of encrypted IP to reconstruct the post-synthesis netlist with some simple Tcl scripting. Furthermore, recent bitstream-to-netlist tools provide a fully automated way to obtain a plaintext netlist of encrypted IP [3]–[8]. These vulnerabilities are discussed in more detail in Section 2.1, and illustrated in Figure 1a.

In this work we explore a fundamentally different approach to IP protection. Rather than allowing the CAD tools to decrypt the IP, we propose a method where the full details of the IP remain hidden from the CAD tools, preventing IP theft from the attacks discussed above. This novel approach centers around *partially* encrypting IP netlists in such a way that allows the CAD tools can still operate on the netlist, while still preventing the user from uncovering the full details of the IP netlist. The IP remains partially encrypted in the bitstream, and is only decrypted during FPGA configuration. This approach is illustrated in Figure 1b.

1.1 Outline

This paper begins with Section 2 defining the threat model addressed by this work, with additional details on vulnerabilities of existing IP protection methods, and bitstream encryption approaches. Section 3 then discusses related work in the field of IP protection.

Section 4 presents our general approach of partially encrypting IP through the CAD flow. This includes a discussion on trade-offs between how aggressively the IP is partially encrypted, with the ease of creating CAD tools capable of operating on the partially encrypted IP. The section also discusses approaches for decryption during FPGA configuration, including how to keep decryption keys hidden from the user. This section also includes a discussion on motivating deployment scenarios where the proposed approach would be useful.

Section 5 presents a proof-of-concept tool that demonstrates one possible implementation of our proposed IP encryption framework. Our implementation provides a multi-vendor solution (meaning IP from different vendors can be encrypted by different vendor keys), works with an existing commercial FPGA tool (Xilinx Vivado), and can be deployed using an existing FPGA device family (Xilinx 7-series). Section 6 provides experimental results of our proof-of-concept tool, detailing resource overheads, CAD and configuration runtimes, and a verification strategy.

Preventing the CAD tools from having full observability of the IP provides strong security benefits, but also comes with notable limitations to the tools available to the IP customer. We discuss the limitations and considerations of our approach in Section 7. Despite these limitations, we maintain that our proposed approach still provides substantially better IP protection than current methods, and should be considered for security-sensitive organizations.

1.2 Contributions

The key contributions of this work are:

- The novel strategy of partially encrypting IP from netlist to bitstream, using techniques where the CAD tools can still operate on the netlist.
- A proof-of-concept implementation that demonstrates how this can be done with existing FPGA tools and devices, including overcoming major challenges, such as:
  - how to perform per-IP decryption using an existing FPGA device,
  - how decryption keys can be kept hidden from the FPGA user, despite them having access to the configuration circuitry and the physical FPGA, and
  - dealing with optimizations performed by the CAD tool on the encrypted netlist.
- Experimental results documenting overhead costs for this approach, including impact on resource usage, CAD runtime and configuration runtime.

2 THREAT MODEL

In this work we are assuming an attacker has access to encrypted IP, and is seeking to obtain the original IP RTL, or the post-synthesis IP netlist. This attacker may be a designer at an organization that has purchased or licensed the IP for use in their product, and is seeking to reverse-engineer

the IP for their own use. Throughout this paper we refer to this potential attacker as the *IP User* or *IP Customer*. The end goal of this work is to prevent the attacker from being able to obtain the full details of the IP netlist, which they could then use to reverse-engineer the IP, modify and/or re-sell the IP, or gain secret internal details, such as trade secrets, algorithmic details, or cryptographic keys.

To clarify, this work is not focused on protecting the *deployed* bitstream from a user of the production system or product. Existing bitstream encryption methods are used to protect against such an attack, which is a distinct problem from the one addressed by this work. To reiterate, we are focusing on protecting the full IP details from a user that has legitimate access to use the IP in their design.

2.1 IP Encryption and Extraction Methods

Current FPGA vendors support IP encryption, following the IEEE-1735-2014 standard. This standard allows IP vendors to encrypt their IP, and distribute it to users of the CAD tool. The CAD tool contains the necessary cryptographic keys in order to decrypt and operate on the IP. The tools perform due diligence to ensure that the tool user is able to incorporate the encrypted IP into their design, and compile the design to bitstream, without ever having access to the unencrypted IP. The user is prevented from inspecting the IP netlist, and is only able to view the IP as a black box. While the tools prevent the user from viewing the IP netlist, the decrypted IP details are still available internally, allowing for full CAD optimizations, simulation, and bitstream generation.

In this paper we detail our proof-of-concept tool that is implemented with Xilinx FPGAs and the Xilinx Vivado CAD tool suite. Given this, some of the details and terminology we use are specific to Xilinx devices and tools; however, this is not meant to suggest that these are Xilinx-specific vulnerabilities. Rather, the vulnerabilities we discuss relate to the fact that FPGA CAD is performed on the unencrypted netlist, and that the bitstream is not encrypted. This is common to all FPGA vendors.

Encrypted IP can be provided to the user in two forms: *soft* IP, and *hard* IP. Soft IP is provided as RTL, and is encrypted by the CAD tool during synthesis. It may contain several configuration options to enable or disable features of the IP, change data widths, bus protocols, etc. Many of the IP provided as part of Vivado's IP library fit into this category as they can be configured in the GUI, but the RTL and resulting netlist remain encrypted (in fact, even the Tcl code to manage the logic behind the configuration GUI is encrypted).

Hard IP is provided as a post-synthesis netlist, and cannot be further configured by the IP user. However, the IP is still decrypted by the CAD tools, enabling it to be integrated into the overall design, and enabling cross-boundary optimizations with the user's own RTL.

While the current industry-standard approach makes some effort to hide the IP netlist from the user, we know of three major methods that allow the user to obtain the encrypted IP, described in the next subsections.

2.1.1 Key Recovery Attacks. With the IEEE-1735 approach, the CAD tools contain the necessary decryption keys to decrypt the IP. If an attacker is able to recover these keys from the CAD tools, they would be able to perform a full decryption of the IP. This would include not only obtaining the post-synthesis netlist (ie. hard IP), but also being able to decrypt the original configurable soft IP. Unfortunately, such attacks have already been demonstrated. In [9], Chhotaray et al. discuss cryptographic vulnerabilities in the IEEE-1735 standard, exposing the keys to possible theft. In [2], Speith et al. demonstrate the feasibility of such attacks, and successfully recover IP encryption keys from seven different EDA vendors, including from the Intel, Xilinx and Lattice FPGA tools. The attacks are performed through analysis of the CAD tool's program memory. With these keys, the

authors are able to decrypt, modify, and re-encrypt all protected IP cores. This exposes not only the threat model discussed in this paper (IP theft), but also the possibility of injecting hardware Trojans into encrypted IP.

2.1.2 Tcl-Based Netlist Recovery. In our experience with the Vivado tool (version 2020.2), when viewing an implemented design checkpoint containing encrypted IP, many details of the IP cells are hidden from the user. For example, the cell properties, such as lookup table (LUT) equations, are hidden, preventing the user from viewing the full netlist details of the IP. However, not all details are hidden. Cell and hierarchal names appear to be visible, and in order to view placement and routing information of the design, the user is able to see the types of cells that are instanced, and the interconnections between them. While cell properties are hidden, with some programming effort, additional information can be recovered. For example, the mapping of logical Cell to physical BEL (eg. LUT, FF) is not hidden, and the physical configuration properties (eg. CONFIG.EQN property) of the BELs are not encrypted. With some Tcl scripting, it is possible to utilize this information to mostly reconstruct the netlist of the encrypted IP. This method would only recover the current configuration of the IP (hard IP), not the original configurable RTL (soft IP). In addition, this method is highly implementation-dependent, and different CAD tools may hide different amounts of information.

2.1.3 Bitstream-to-Netlist Tools. While FPGA CAD tools could be patched to make the above attacks more difficult (and perhaps newer versions of the tool already do this), the fact remains that the IP is no longer encrypted in the final bitstream, and a capable user could still obtain the IP netlist from the bitstream. Although bitstream formats have been historically kept proprietary, many modern open-source tools have documented commercial bitstream formats, and even provide automated bitstream-to-netlist tools [3]–[8]. Our experience has been that these tools are quite straightforward to use; new research students in our lab have been able to use them with only a few hours of learning. In fact, the tools are sufficiently accurate that in another project we have been able to utilize them to prove design equivalence between netlists and bitstreams in two different FPGA families [10], [11]. Given the bitstream format is tied to the physical FPGA device architecture, there is no way to simply remove this vulnerability through updating the CAD tools.

However, this approach has limitations. The bitstream contains no hierarchy, signal, or instance names, so the reversed netlist that is produced is a flat, nameless netlist. This makes it much more difficult for an attacker to understand. However, there is also a growing body of research on gaining higher level understanding of such netlists [12]–[16]. In addition, while the attacker would need to separate the IP from the rest of the design logic, it would be trivial for them to generate a design containing only the encrypted IP they were interested in obtaining. Finally, like the previous method, this method would only gain access to a netlist of the IP as it has been configured, not the original configurable RTL code. However, the attacker could reconfigure the IP and repeat the process repeatedly to gain a netlist of any desired configuration of the IP.

2.2 Relevance of Bitstream Encryption

While FPGA vendors offer bitstream encryption technology, this is *not* designed to function as a protection mechanism for third-party IP, and does not address the threat model discussed in this paper. Bitstream encryption is applied by the system designer, and is used to protect a deployed system. It prevents attackers who may gain access to the physical FPGA from obtaining the unencrypted bitstream, and thus the netlist of the entire design. However, the IP customer who is designing the system always has access to the bitstream in unencrypted form, since they are the ones responsible for applying the bitstream encryption. There is currently no mechanism to prevent the IP customer from obtaining the unencrypted bitstream.

3 RELATED WORK

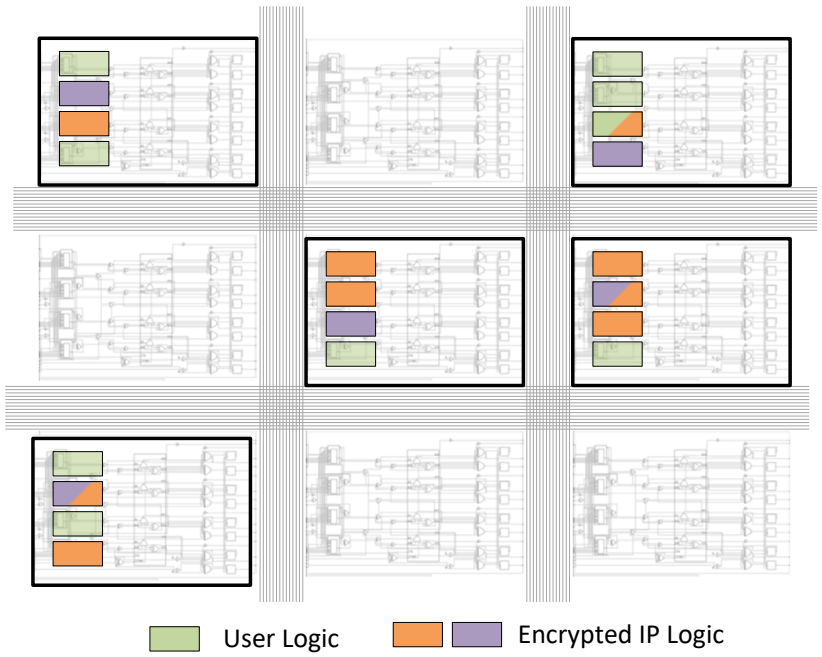
There is some related work in the field of protecting third party IP for FPGAs. Similar to our work, these works focus on preventing the IP customer from being able to view the full details of the IP netlist. Some works also focus on enforcing licensing restrictions on the IP, such as preventing the IP from being used in more than a certain number of designs, or beyond an expiry date of the license.

In [17], Kean discusses an IP protection scheme that also proposes encrypting the IP and generating an encrypted bitstream. However, the paper appears to be a conceptual proposal, and does not contain any discussion or details on encryption schemes for the IP, or how the CAD flow would successfully operate on the encrypted IP and incorporate it into the user's design. In addition, the encrypted bitstream is not designed to be decrypted by the FPGA device, but rather by a trusted computer system that is connected to the internet and connected to the FPGA. This trusted system contains the configuration keys to decrypt the design prior to programming the FPGA. While such a system provides high flexibility (eg could enforce licensing restrictions), deploying such a system in practice may be challenging. In addition, the system would be in control of the attacker, and susceptible to the same key recovery attacks discussed in Section 2.1.

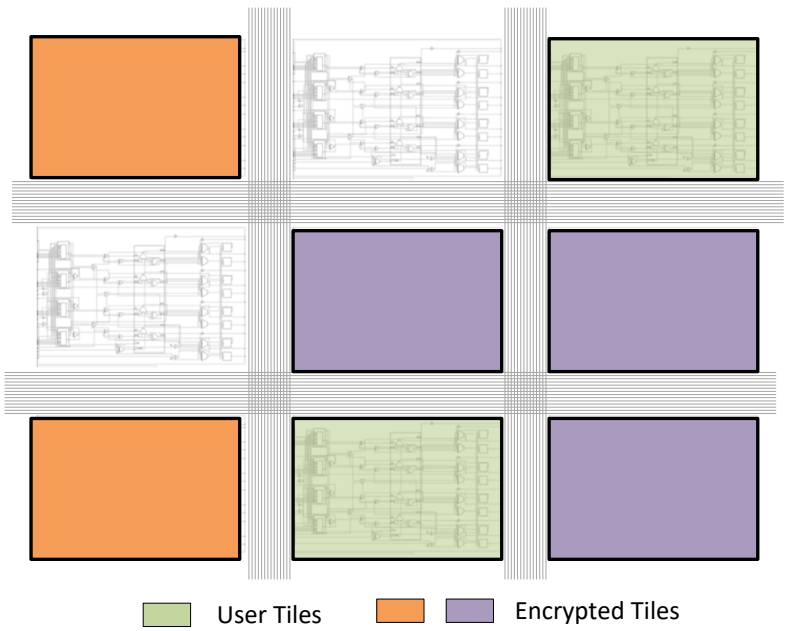
In [18], Gaspar et al. propose an approach that can be used in a multi-FPGA system. In their work, each encrypted IP is used on a separate FPGA, negating the need for CAD tools to operate on the encrypted IP and incorporate it into the user's design. The work focuses on developing a secure boot-up process on each FPGA that is able to retrieve the encrypted IP and configure it to the FPGA, as well as a secure communication scheme between the various FPGAs. While this appears to be an effective approach, the requirement of a separate FPGA per IP may be too restrictive for many applications. In addition, it appears that the IP would need to be pre-compiled for a specific FPGA device by the IP vendor.

In [19], [20], Kepa et al. also focus on protecting the IP from the user through encryption. Their work proposes that IP vendors place and route their design for a specific partial reconfiguration slot on the FPGA. This fully implemented IP can then be encrypted and distributed to the user. The FPGA system contains a Secure Reconfiguration Controller (SeReCon), which is able to decrypt the IP and reconfigure the partial region with the IP. The SeReCon is also designed to enforce licensing restrictions, such as pay-per-use and time-limited licensing. While the SeReCon architecture provides many IP protection features, the requirement of IP being pre-implemented for a specific partial reconfiguration slot may be too restrictive for many applications. In addition, the work does not discuss how IP decryption keys are securely transferred from the IP vendor to the SeReCon trusted controller without interception by the IP user.

In comparison with these works, our work focuses on a substantially different approach where IP does not need to be pre-implemented for a specific FPGA device or region, but can be compiled into the user's design using existing design methods. This provides a much more user-friendly design experience, and is a similar design experience to existing IP encryption methods used by the commercial CAD tools. This flexibility comes at a cost, as our approach requires only partially encrypting the IP. Depending on how aggressively the framework encrypts the IP, cell types and interconnect patterns may still be visible to the user. However, in the worst case, we are not exposing any more information than is already readily visible when inspecting encrypted IP in current FPGA CAD tools, while ensuring that the encrypted IP data is not vulnerable to the attacks discussed in Section 2.1.



(a) Cell-level encryption, specifically encrypting LUT logic functions



(b) Tile-level encryption

Fig. 2. Alternative approaches for partially encrypted FPGA IP. In this paper we demonstrate a proof-of-concept of the LUT-level encryption approach.

Intertile connectivity could be obfuscated by adding extra fake nets (ie nets have multiple drives), that aren't actually enabled on FPGA programming.

4 APPROACHES FOR PARTIALLY ENCRYPTED IP

In this section we describe our vision for an IP protection solution, which would support partial encryption of IP, from netlist to bitstream. Later, in Sections 5 and 6 we present and evaluate our proof-of-concept tool, which provides one possible implementation of these ideas, and is designed to work with existing FPGA devices and CAD tools. However, in this section we provide a broader description of our vision, independent of any specific implementation, or current FPGA devices. Ideally, a solution supporting IP encryption would have the following properties:

(1) Netlist remains (partially) encrypted during the entire CAD flow, including placement, routing, and bitstream generation.
(2) Support for multiple keys, allowing different IP vendors to encrypt their netlists with different keys.
(3) The FPGA device can support a bitstream with per-IP encryption, and decrypt the design during configuration.

These goals and requirements are described further in the following subsections.

4.1 Encryption during CAD flow

While ideally a third party would be able to provide a user with a completely encrypted version of their IP, this would prevent the CAD tools from being able to operate on the design. Even for a fully synthesized and technology mapped design, the CAD tools still need to be able to perform packing, placement, routing, and bitstream generation. Thus, the IP must be encrypted in a way that allows the CAD tools to operate on the design, while still preventing the user from being able to uncover the full details of the IP.

4.1.1 Cell-Level Encryption. One approach to this problem is for the IP vendor to synthesize their netlist, and then partially encrypt it. For example, the logic functions of the LUTs could be encrypted, while the routing between the LUTs is left unencrypted. Other cells could also have properties encrypted, such as RAM or flip-flop initialization values. With this information the CAD tools could still perform packing, placement, and routing. Even bitstream generation could be performed, provided that the properties could remain encrypted in the generated bitstream.

Figure 2a illustrates the concept of partially encrypting LUT values, and shows Xilinx-like slices where LUTs maybe be a mixture of encrypted and unencrypted. Once challenge that can be encountered is that modern LUTs are fracturable, allowing potentially two logic functions to be mapped to the same physical LUT. As we demonstrate later in our proof-of-concept tool, as long as this is accounted for, it is not an issue, and in fact the same physical LUT can be shared by different IP encrypted with different keys.

Extra care must be taken to ensure that once the IP vendor has synthesized and encrypted their netlist, it remain largely unchanged through the CAD flow. We show in our proof-of-concept tool that certain physical optimizations can be accounted for (eg. reordering LUT inputs); however, other more aggressive optimizations, such as re-optimizing the logic across LUTs, or cross-boundary optimizations between IP, may not be possible to handle.

This approach of encrypting cell properties may not be possible with every commercial FPGA CAD flow, and may require a custom CAD tool to achieve; however, in Section 5 we show how it is at least possible to achieve this with the Xilinx Vivado tool.

4.1.2 Tile-Level Encryption. While encrypting logic functions hides some details of the IP, exposing the full connectivity pattern of the IP may not be desireable.

Another approach would be to have the IP vendor perform packing of their post-synthesis netlist into tiles, along with intra-site routing, and then encrypt the entire tile. This approach is illustrated

in Figure 2b. Since many nets in a design are absorbed into routing internal to tiles, the connectivity of these nets would be encrypted and hidden from the user.

While tile-level encryption would be better at hiding the details of the IP, it comes with several drawbacks:

- LUTs from different IP would no longer be able to share the same physical LUT, or even the same tile, likely degrading quality of results (QoR). The tile usage would likely increase due to to fragmentation, and critical path may increase as a result. If the critical path crossed IP boundaries, the QoR degradation would be even more significant.
- Certain physical optimizations would be prevented, such as reordering LUT inputs to ease routing congestion. This would likely also lead to worse QoR.
- Finally, such an approach would also be much harder to achieve with a commercial CAD tool. While fake tile information could be provided to the commercial CAD tool, it may not be possible to prevent some of the physical optimizations that would break the approach. A custom CAD tool could be used to support this approach, but of course this would substantially increase the development effort.

4.1.3 Beyond Tile-Level Encryption. If security warranted even more aggressive optimization beyond tile-level encryption, it may be possible, but would likely require even more drastic CAD tool support and QoR cost. For example, one could encrypt multiple tiles together along with routing channels between these tiles; however, it would likely prevent any other routes from passing through these channels. Alternatively an encrypted IP could contain fake connections to better obscure the connectivity pattern, which could be removed at decryption and configuration time, at the expense of increased routing congestion. However, we believe that such drastic approaches would likely be too expensive and complicated to implement. At that point it would likely be better to resort to having encrypted IP be pre-implemented for entire partially reconfigurable regions, as has been done in some previous works [19], [20].

In our proof-of-concept tool described in Section 5, we demonstrate the feasibility of the first approach, cell-level encryption. While tile-level encryption would be interesting future work, it would require more drastic CAD tool changes.

4.2 Support for Multiple Keys

Ideally a good framework for IP encryption would allow for different IP to be encrypted with different encryption keys. This would allow a user to obtain encrypted IP from a variety of sources, and would prevent these IP providers from being able to view each other's IP.

4.3 FPGA Configuration and Decryption

While generating a bitstream with encrypted IP is useful, this is only half of the challenge. The FPGA device must also be able to decrypt the IP during configuration. The decryption circuitry must contain the various encryption keys, and metadata within the bitstream would need to indicate which parts of the bitstream contents are encrypted by which vendor. In addition, it is not sufficient to simply encrypt certain bits of the bitstream, as physical optimizations performed by the CAD tool necessitate providing extra metadata along with the traditional bitstream bits. For example, in our proof-of-concept tool, we need to capture and include optimizations such as LUT input reordering, fracturable LUT combining, and other optimizations in the included metadata (more details are provided later in the paper).

While current FPGAs typically support encrypted bitstreams, it is expected that the entire bitstream is encrypted with a single key, and the decryption and configuration circuitry does not support selectively decrypting different parts of the design with different keys. In addition, there is

no logic to receive the previously mentioned metadata of physical transformations, and update the bitstream appropriately.

Given these limitations, we identify two possible approaches to support decryption of IP during configuration:

(1) **New FPGA Devices:** One possible solutions is to design new FPGA devices where the configuration circuitry is enhanced to support decryption of IP. While we do not explore this approach in detail in this paper, we provide a brief discussion in the next subsection.
(2) **A Static "Loader" Shell:** An alternative approach that uses existing FPGA devices is to build the required configuration circuitry into static logic in the FPGA fabric, restricting the user's design to a subset of the device. This approach is used in our proof-of-concept tool, and discussed in the subsequent subsection.

4.3.1 Enhancing FPGA Architecture to Support Encrypted IP. The ideal solution to configuring bitstreams with encrypted IP would be to have all of the required functionality built into the configuration circuity of the FPGA itself. This would require the following modifications to the FPGA architecture:

(1) **Fine-gained decryption using multiple keys:** Most FPGAs already contain a mechanism for decrypting the bitstream during configuration, and contain hardware-accelerated decryption. We believe this circuitry could largely be reused, with some modifications to choose which key to use for decryption for different parts of the bitstream. Although the area overhead of the changes may be small, the runtime overhead may be more significant. Depending on the encryption scheme used, the interval at which the encryption needs to switch keys may vary. For cell-based encryption, which is very fine-grained, the key may need to change quite often, which would likely add latency to the configuration process.
(2) **Transformation circuitry:** Additional circuitry would be required to accommodate transformations performed on the encrypted regions by the CAD tool post-synthesis. For example, in our proof-of-concept tool, we detect when LUT inputs are re-ordered by the CAD tool during implementation, and add this reordering information to metadata that is sent to the FPGA along with the bitstream. After the original logic function (which was encrypted post-synthesis before any input reordering) is decrypted by the configuration logic, it then needs to use this metadata to update the logic function accordingly. There are other similar transformations that we discuss later in the paper. Handling these transformations would require additional circuitry in the FPGA configuration logic, although for some modern FPGAs that contain dedicated configuration processors, this may be possible to implement in software.
(3) **Key storage:** The FPGA device would need to contain a mechanism for inputting and storing multiple keys. Many current devices use eFuses to store the bitstream decryption key, which is a write-only, write-once mechanism. This same mechanism could be expanded to support multiple keys.

    The main difference is that with current technologies it is usually the system designer that is loading the key onto the FPGA eFuses; however, with our proposed approach, the IP vendors would be loading the keys. The write-only nature of the eFuses would ensure that the user could not read the keys back from the FPGA. However, one would still need a way for the IP vendor to physically perform the programming. This may necessitate the IP vendor having physical access to the FPGA device, or having a trusted third party perform the programming. We discuss some scenarios in Section 4.4 where this may be practical.

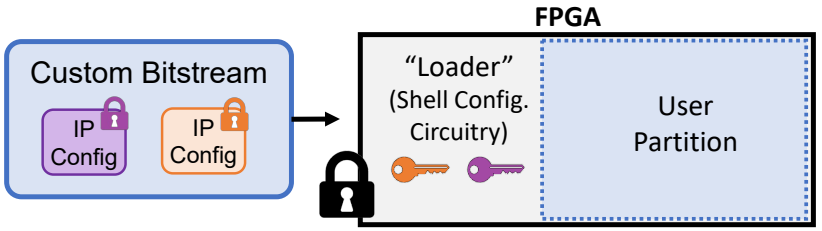


Fig. 3. Loader Shell

4.3.2 A Static "Loader Shell". While we advocate for adding the above features to the configuration circuitry of future FPGA devices, we also explore the more practical scenario of targeting existing devices. The additional configuration features described above can be implemented in a static region of FPGA logic, which we refer to as a "Loader Shell", with the user's design being loaded into a reconfigurable partition of the FPGA via partial reconfiguration. This concept is illustrated in Figure 3. The loader design contains the functionality to receive the custom bitstream over a communication channel (eg. UART, USB), decrypt and patch the bitstream, and configure the user partition.

The loader design must contain the decryption keys, and as such, must be kept hidden from the user. This can be achieved by leveraging existing FPGA bitstream encryption features, which allows for the entire loader design to be encrypted and unreadable by the user. The loader shell, along with the decryption keys, would need to be maintained by a trusted third party.

While the Loader shell approach allows our encrypted IP framework to be deployed on existing FPGA devices, the restriction of the user logic to a partition, as well as the need for a trusted third party are notable disadvantages versus a custom FPGA device with enhanced configuration circuitry.

In addition to the broad requirements mentioned here, several subtle implementation details must be carefully considered in order to ensure that the keys stored in the loader shell cannot be easily recovered by the user. Our proof-of-concept Loader Shell discussed in Section 5.3 discusses many of these details.

4.4 Motivating Deployment Scenarios

While we believe our approach for IP encryption is effective at protecting IP details, we recognize that the effort required to deploy such a solution may be substantial, and the limitations (which we discuss later in Section 7) are not trivial. As such, we do not expect that this would be used in all design scenarios. However, as motivation, we describe a couple scenarios where our proposed framework would work well.

(1) *Government/Defense IP Repository:* In the case of a large state government, there may be several organizations that create IP and may want to share and collaborate with other organizations. To limit exposure of sensitive designs to fewer parties, it may be desireable to encrypt IP before distribution. In addition, if the IP were to contain cryptographic keys (for example, in communications systems), it may be necessary to allow other organizations to use the IP in their systems without disclosing the underlying keys. In this scenario, each organization may have their own key to encrypt their IP, with a central organization maintaining the keys and programming them into the FPGA devices.
(2) *Cloud service provider (CSP):* An FPGA cloud provider may want to distribute useful IP to their customers in order to make their cloud offerings more attractive, but may not want

to expose the details of the IP. In this case, the CSP could encrypt the IP with their own key, and distribute the IP to their customers. FPGA cloud providers already operate with a shell/user region model, where the CSP provides a shell design that is loaded into the FPGA, and the user places their design in a reconfigurable region. In this case it would be straightforward to add the Loader logic to the shell, along with the keys for the CSP's IP. A shell with the decryption keys stripped out can be used by the customer to implement their design. Given that the CSP maintains ownership of both the FPGA and the shell design that contains the decryption keys, there is very minimal risk of IP theft.

5 PROOF-OF-CONCEPT TOOL

This chapter describes our proof-of-concept flow that partially encrypts IP all the way through *placement*, *routing*, and *bitstream generation*. It demonstrates, beyond basic plausibility, how to deal with challenges such as the optimizations the CAD tools make on the encrypted design, how to decrypt the vendor IP, and how to keep the encryption keys hidden from the user.

Our general approach is to remove and encrypt LUT *INIT* properties in the protected IP, thereby hiding significant internal details of the IP, while still allowing a commercial CAD tool to operate on the IP. The produced bitstream will not contain the correct LUT *INIT* values, so intelligent configuration logic is required to decrypt LUT *INIT* values and correct them on a per-LUT basis. Our tool consists of three main components:

(1) **Vendor Flow:** Allows the IP designer to partially encrypt the IP, as described in Section 5.1.
(2) **User Flow:** Allows the user to instantiate one or more encrypted IP in their design, and generate a partially encrypted bitstream. This is detailed in Section 5.2.
(3) **Loader:** A static region that contains the decryption and configuration circuitry, and hides the decryption keys from the FPGA user. Described in Section 5.3.

5.1 Vendor Flow: IP Encryption

The IP vendor flow is shown in Figure 4, and involves the IP vendor first performing standard Vivado Synthesis on their RTL to produce a synthesized design checkpoint, which contains the design netlist. The synthesis is performed out-of-context, and with full hierarchy flattening enabled. The vendor then runs a provided Python script that takes the design checkpoint, along with a vendor-selected AES encryption key. The script then performs the following steps:

- RapidWright [21] is used to load the design checkpoint and locate all LUTs in the design.
- For each LUT, the script:
  - encrypts the *INIT* property, padded with a random nonce to prevent identical LUTs from having the same ciphertext,
  - writes out the unique hierarchal LUT name and *INIT* ciphertext to a file, and
  - modifies the original *INIT* to be a logical-AND of all the LUT inputs. This keeps any

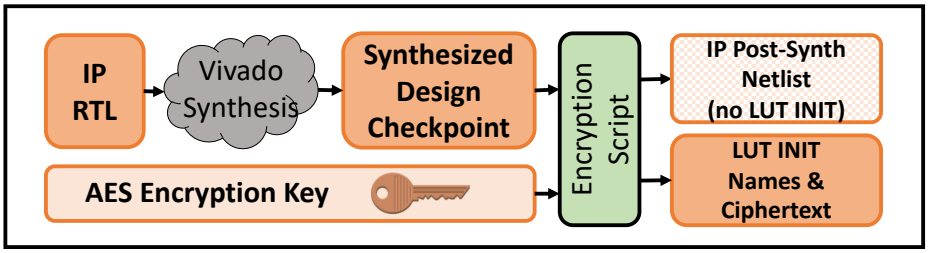


Fig. 4. IP Vendor Flow

Note: It is not sufficient to simply encrypt the *INIT* properties in place. As the CAD tools perform implementation of the IP, they may re-order LUT inputs, causing a shuffling of the *INIT* bits. It is not valid to perform this shuffling on the ciphertext, as it would corrupt the data. Instead, the ciphertext must be kept separate and untouched.

- The new netlist is written out to a Verilog file (all *INIT* properties have now been effectively removed), and the `DONT_TOUCH` property is applied to the module via in-code pragma. This way, wherever the netlist is instantiated by the user, the module hierarchy and LUT names will remain the same, and not merged with other logic. An example snippet from an encrypted netlist in shown in Listing 1.

```
(* DONT_TOUCH = "yes" *)
module des3_perf (
    desOut,
    desIn,
    decrypt,
    key,
    clk
);
...
    LUT3 #(
    .INIT(8'h0) # INIT replaced to AND
    ) LUT_instance_name (
        .I0(signal1),
        .I1(signal2),
        .I2(signal3),
        .O(signal4)
    );
...
endmodule
```

Listing 1. Code snippet illustrating a portion of an encrypted IP netlist.

The final produced netlist, along with the LUT names/ciphertext file, is provided to the IP user. Although our tool does not rename and anonymize LUT names, it would be trivial to do so, and would remove any details of the IP that the user could obtain by knowing internal instance names.

In our tool, each LUT is encrypted separately. This is done because our Loader system (described later in Section 5.3), performs configuration in a streaming fashion, and does not have enough memory to receive the entire user bitstream. As such, when the LUTs are decrypted in the device, they need to be decrypted individually, potentially in a different order than they were encrypted.

5.2 User Flow

The user flow is shown in Figure 5, and details the steps required for an IP customer to instantiate encrypted IP in their design, and generate a special bitstream that can be loaded onto the FPGA.

The encrypted IP netlist created as described previously is still a valid Verilog netlist (albeit LUT *INIT* properties are meaningless); as such, the user can instance it in their design in the same they would with regular IP.

Since our Loader Shell architecture requires the user to target a partial region of the FPGA, the user would follow a standard partial reconfiguration design flow. This involves the user synthesizing

their RTL to produce a design checkpoint, which is then assigned to the reconfigurable partition. The static portion of the FPGA (the Loader Shell) is provided to the user as a pre-implemented design checkpoint. This Loader Shell is identical to the one that will be present on the deployed FPGA, except that the encryption keys have been removed from memory. Standard implementation and bitgen can then be run by the user to produce a partial bitstream. The *INIT* bits in the bitstream corresponding to encrypted IP will still be invalid and meaningless at this point, meaning that even with bitstream-to-netlist reverse engineering tools, the user cannot obtain the protected IP.

It should be noted that although the user is targeting a reconfigurable partition, there is no restriction to place separate IP in different partitions, like in [19], [20]. Rather, the partition is one large region that contains all of the user's design, and would be made as large as possible on the device to maximize the resources available to the user.

Next, the user runs a provided Python script, providing the LUT *INIT* ciphertext they received from the IP vendor, along with the implementation checkpoint and partial bitstream produced by Vivado. The script uses RapidWright [21] to read the implementation checkpoint, and locate all encrypted LUTs based on their hierarchal name provided in the ciphertext file. The script then uses RapidWright to generate a collection of implementation metadata that is packaged along with the partial bitstream and ciphertext. The metadata is required for the Loader to patch the bitstream at configuration time, and contains the following for each encrypted LUT instance:

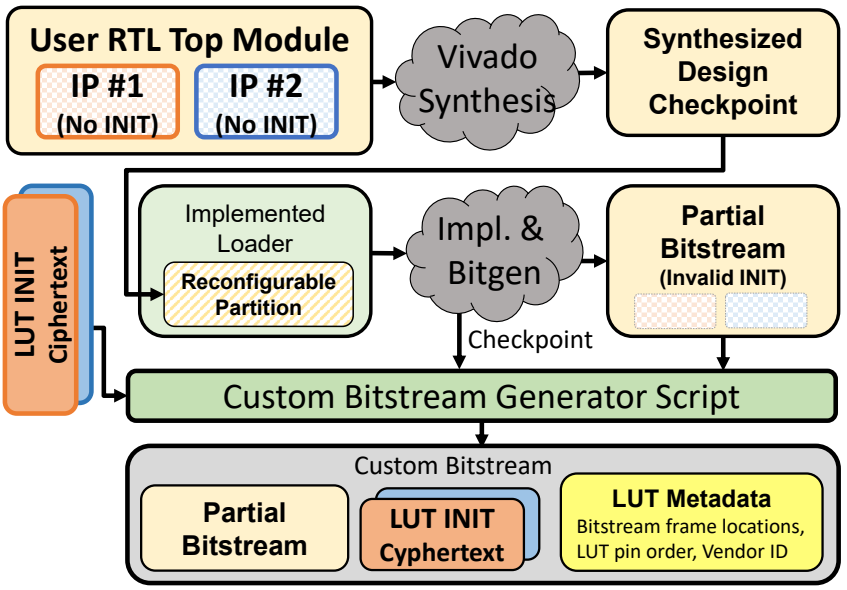


Fig. 5. User Flow, where encrypted IP is instantiated in the user's design, and a partially encrypted bitstream is generated.

(3) **LUT Sharing Information:** The Xilinx 7-series LUT primitive (LUT6_2) can implement a six-input logic function, or two five-input (or smaller) logic functions provided there are five or fewer unique input signals. Given this LUT fracturability, a logical LUT can be assigned to either half (O6 or O5 output) of the LUT6_2 primitive. This affects whether the LUT utilizes the entire 64-bit *INIT* value, or just the lower or upper 32 bits. Special care must be taken here since the LUT6_2 primitive could be shared with other LUTs from the same encrypted IP, a different encrypted IP, or even non-encrypted user logic.
(4) **Physical Location:** The tile location of the LUT, along with the base address and word offset of the LUT's configuration bits within the bitstream. This information is gathered using the *tilegrid.json* file from Project X-Ray [3] for our specific FPGA part.

In most cases, the logical LUTs from the netlist can be found in the implemented design using RapidWright to do simple name matching. However, certain special cases arise due to optimizations performed by the CAD tool, that require extra care. One such case arose due to *LUT routethrus*, which are LUTs that are used by Vivado in the implemented design to route a net to a flip-flip, but are not reported as logical LUTs in the netlist. However, they still need to be accounted for in the produced metadata as they may share a LUT6_2 with an encrypted LUT, and care must be taken to not overwrite the half of the *INIT* property that contains the routethru logic.

A similar challenge can occur with *constant generating LUTs*, which are LUTs that Vivado inserts in the implemented design to generate a constant value when routing to a traditional constant-generating primitive may be difficult. Again, care must be taken to include this information in the metadata to ensure the LUT behavior is preserved.

The final output of the Python script is a custom bitstream shown in Figure 6. The standard bitstream frames are augmented with additional information, and each frame indicates whether it can be directly configured to the FPGA (*Write Only*), or whether the frame contains encrypted LUT metadata (*Write Edit*), in which case the frame data is augmented with the necessary ciphertext and metadata discussed above.

5.3 Loader Shell

Once the user has created the modified bitstream containing the metadata necessary for patching, the Loader Shell can be used to load the bitstream onto the FPGA. In our implementation we chose to use a MicroBlaze soft processor to handle the decryption and bitstream patching, as implementing the functionality in software is much faster to prototype than in hardware. However, the performance of our Loader is quite slow, and in a real deployment, the decryption and patching could likely be implemented much faster with custom hardware.

Our Loader Shell architecture is shown in Figure 7, and contains a MicroBlaze processor, a UART core to receive the bitstream from the user, the decryption keys stored in BRAM, and interface logic to the Internal Configuration Access Port (ICAP). We tested our system on a Nexys4DDR development board, containing an Artix-7 100T FPGA.

5.3.1 Configuration Process. Our configuration process proceeds by having a script on the host computer send portions of the bitstream to the FPGA over UART. The MicroBlaze memory is limited, and it cannot receive and store the entire bitstream in one transmission, so the bitstream must be send in pieces, with each piece processed and configured before the next is sent. Bitstream data for Xilinx FPGAs is broken into *frames*, which for the Artix-7 family are 101 32-bit words in length. While it may seem logical to send one frame at a time, this is actually not possible as each LUT's 64 INIT bits are spread across four different frames, and we need to be able to decrypt and patch the entire LUT's INIT bits at once. We actually elect to send 36 frames (14,544 bytes of frame

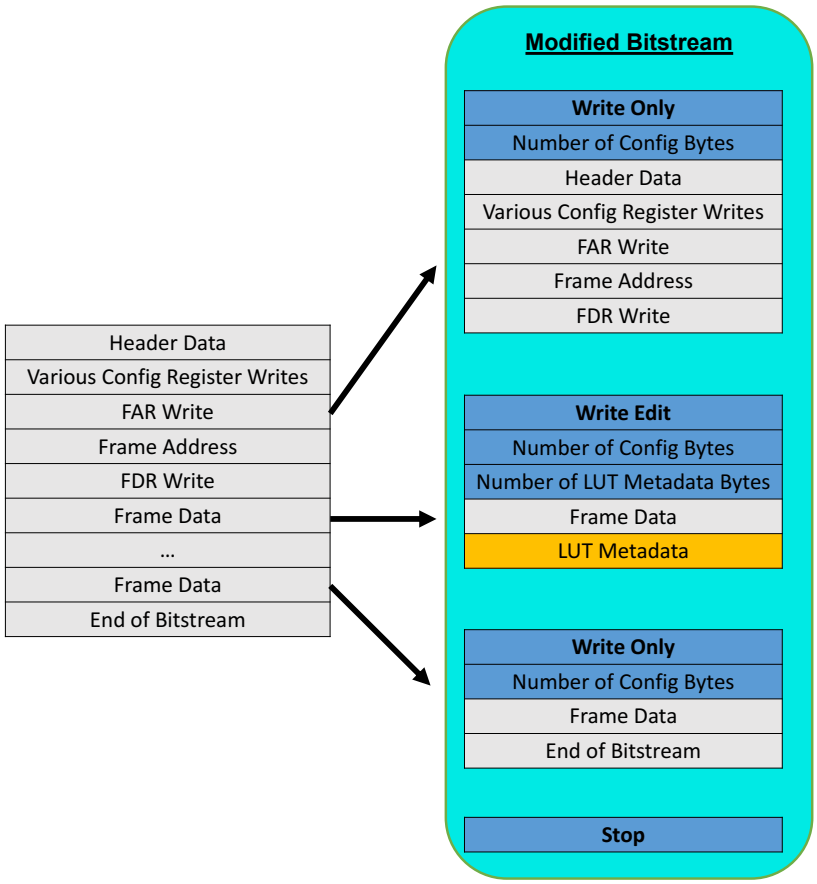


Fig. 6. Custom bitstream format that contains a mixture of standard bitstream frames, and custom frames that contain encrypted LUT metadata.

data plus the metadata shown in Figure 6) at a time, as this represents all of the frame data for a CLBLL tile.

The loader receives the frames into memory, and must first verify that the frame locations belong to the region of the FPGA allocated to user logic. This is necessary to prevent the user from overwriting any Loader logic. Next, the Loader software iterates through all LUT metadata objects, and:

- Decrypts the *INIT* value and discards the nonce, using the vendor ID to choose which key to decrypt it with.
- Inflates the *INIT* value to 32 or 64 bits (depending on whether the LUT is shared). This is necessary because the LUTs in the post-synthesis netlist that were originally encrypted may have had fewer inputs (eg. *LUT2*, *LUT3*, etc.), and thus fewer *INIT* bits. The *INIT* value must be inflated to the correct number of bits before it can be patched into the bitstream. In a standard unencrypted IP flow, an IP instanced in a user's design would have INIT values inflated during implementation of the user's design. This is not possible in our encrypted flow, as there is no way to apply this inflation transformation on encrypted data. As such,

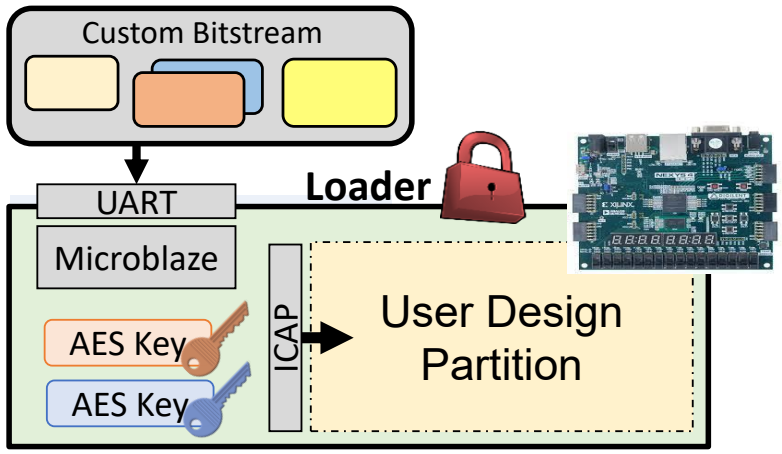


Fig. 7. Loader architecture, which receives the encrypted bitstream from the user, decrypts the *INIT* values, and patches the bitstream before writing it to the ICAP.

the INIT inflation must take place during decryption on the device. Technically, in our flow, Vivado will inflate the INIT bits of the AND gate we used as a placeholder; however, this data is meaningless and discarded by the loader.

- Reorders the *INIT* value based on the logical to physical pin mapping.
- Checks whether another LUT from the netlist shares the same physical LUT. If so, this second LUT is also processed, and the two resulting 32 bit *INIT* values are combined into the two halves of the 64-bit *INIT* value for the physical LUT.
- Writes the now correct *INIT* value into the appropriate bit locations in the bitstream. The locations are determined using the data provided by the `segbits_clbll_l.db` or `segbits_clblm_l.db` files in Project X-Ray [3]. This differs for CLBLLs and CLBLMs so the tile number field is used to indicate which bits to write to specifically.

When all LUT metadata objects have been processed, the patched 36 frames of configuration data are written to the ICAP.

It is worth noting that if the Loader functionality were to be implemented in hardware as part of the FPGA configuration circuitry, the above transformation steps would need to be implemented in hardware as well. This would add additional hardware costs that may be prohibitive, depending on the application. Unfortunately we have not yet explored a hardware implementation of the Loader, and as such, do not have a good estimate of the area overhead.

5.3.2 Key Management. Perhaps the most challenging portion of this project was ensuring that the decryption keys stored in the loader design remain hidden and inaccessible to the IP user. This would be easier to accomplish if the FPGA was designed with this requirement in mind; however, accomplishing this objective on an existing FPGA device is more challenging.

Our solution to this problem is to use the following approach:

- The encryption keys are stored within the software that runs on the MicroBlaze (BRAM memory in the static Loader bitstream). This BRAM memory is *not* initialized in the implemented loader design checkpoint that is provided to the user as part of the user flow shown in Figure 5. Rather, the software is part of the static Loader bitstream that is provided to the user for device configuration.

- The static Loader bitstream, which contains the decryption keys in its BRAM, is encrypted using traditional bitstream encryption methods. The key for this bitstream encryption is *not* provided to the user, or to the IP vendors. Rather, in our proof-of-concept tool we rely on a trusted third-party key holder which would distribute this encrypted Loader bitstream.
- In order for the Loader bitstream to be configured onto the FPGA (when the user does not know the encryption key), the key must be pre-programmed into the appropriate FPGA eFUSE register, again by the trusted key holder. The user may send their FPGA to the trusted key holder, or they may obtain their FPGA from the key holder, who could serve as the hardware distributor.

In Section 4.4 we discussed two motivating scenarios where our proposed framework would work well. In the first case, where this framework was used by a government defense organization, a single government agency would need to serve as the trusted key holder. This agency would be responsible for creating the trusted Loader design, embedding the various keys into the Loader, encrypting the Loader bitstream, and pre-setting the FPGA eFUSEs to accept the Loader bitstream. This agency would then serve as the distributer of FPGA devices to other government organizations or defense contractors, along with providing organization-specific encryption keys that each organization could use to encrypt their own IP. This would provide compartmentalization between different government organizations and contractors, following the principle of least privilege (PoLP).

The other motivating example in Section 4.4 was a cloud service provider (CSP) that could encrypt their IP and provide it to customers for embedding into their designs. In this case there would only need to be a single encryption key (belonging to the CSP), and the CSP would distribute their encrypted IP to the client. There would be no need to encrypt the Loader bitstream or program the FPGA eFUSE registers, as the CSP would maintain full control of Loader static region, and the FPGA hardware.

The CSP could elect to support encrypted IP from third parties; however, the CSP would need to serve as the trusted key holder, and be privy to the key and full IP details of the third-party IP. This would be in the best interest of the CSP, since they typically have strict requirements on scanning the bitstream for any malicious content before deploying on their systems, and it would not be possible to detect, for example, power wasting attacks [22], [23], without viewing unencrypted LUT values.

In both of these scenarios, the trusted key holder would be capable of creating IP encrypted with any of the various organization-specific keys. While it may be possible to prevent this with a public-private key system, this would need to be properly explored in future work.

5.3.3 eFUSE Settings. It should be noted that the hardware distributor would also set the following eFUSE values to eliminate any method of the user accessing the Loader design and decryption keys:

CFG_AES_Only: Forces the use of AES key stored in eFUSE and disables device readback. This bit must be set as the FPGA must only accept the Loader bitstream, and not another bitstream created by the user to access the IP decryption keys.

AES_Exclusive: Disables partial reconfiguration from external configuration interfaces but still allows partial reconfiguration via the ICAP. This bit must also be set for the same safety concern as the previous bit.

W_EN_B_Key_User: Disables programming of AES key. This bit must also be set to prevent the IP user from overwriting the AES key.

R_EN_B_Key, R_EN_B_User: Disables reading and reprogramming of AES key.

W_EN_B_Cntl: Disables any further changes to the eFUSE registers.

We note that we were fortunate that the Xilinx 7-series devices supported the above functionality in their eFUSE registers. In particular, we rely on the ability to continue to perform internal partial reconfiguration via ICAP with unencrypted partial bitstreams, while disabling external partial reconfiguration via JTAG, and forcing external configuration to only accept a single encrypted bitstream. We are not sure how widespread this functionality is on other FPGA device families and vendors, so our proof-of-concept may be limited in which devices it can be deployed on.

6 EXPERIMENTAL RESULTS

In this section we describe the evaluation of our proof-of-concept tool, which includes both an analysis of the costs associated with the technique, as well as a discussion of our verification methodology to ensure the design is correctly implemented.

We evaluated our proof-of-concept tool using a collection of benchmark circuits, measuring impact on resource utilization, CAD runtime, and configuration runtime.

6.1 Benchmarks

Our testing is performed using nine different user circuits, comprised of 24 different IP obtained from Open Cores. The top-level circuits are synthetic designs that stitch together the Open Cores IP to create a larger design. To stress our design flow, and obtain results in the worst-case scenario, *all* of these IP are treated as encrypted IP, leaving very little unencrypted user logic at the top-level. The circuit details are included in Table 1, which show that the top-level designs range from containing one to five different IP, including some cases where multiple instances of the same IP are used.

6.2 Resource Overheads

In our proposed flow, there are two sources of resource overhead:

(1) By encrypting the IP and marking them as `DONT_TOUCH`, the CAD tools are prevented from performing optimizations across the boundary of the encrypted IP. This overhead will be present regardless of whether configuration is done using a custom FPGA, or using the Loader Shell approach. This is analyzed in Section 6.2.1.
(2) When using the Loader Shell approach, the user design is restricted to a smaller portion of the FPGA, as the Loader Shell must be placed in a static region. This overhead is discussed in Section 6.2.2.

6.2.1 Resource Overhead of Encrypted IP. Table 1 provides results detailing the resource overheads due to encrypting the IP. The left side fo the table, *Regular CAD Flow*, shows the resource usage of the design when running a standard Vivado flow on the entire RTL design, without encrypting any IP. The right side of the table, *Encrypted IP CAD Flow*, shows the resource usage of the design when using our proposed flow, and encrypting all IP. The LUT and FF overhead is included in the right-most columns (no change in RAMBs or DSPs were observed).

The table includes resource breakdown between the IP cores; however, it should be noted that when cross-boundary optimizations are enabled (such as in the *Regular CAD Flow*), the values reported by Vivado can be very skewed between the IP. As such, we do not directly compare the resource usage of the IP cores between the two flows, but rather focus on the overall resource usage of the design.

The overhead of encrypting the IP ranged from -1.3% to 13.7% increase for LUTs, and 0.0%–3.3% increase for FFs.

6.2.2 Resource Overhead of Loader Shell. Our test system consists of a Digilent Nexys4DDR development board, containing a Xilinx Artix-7 100T FPGA. This FPGA consists of eight clock

Table 1. Resource overhead on user's design when using encrypted IP

Design	Regular CAD Flow			Encrypted IP CAD Flow*			Overhead*	
	LUTs	FFs	R/D	LUTs	FFs	R/D	LUTs	FFs
Synth1 (top)	2629 (4%)	1421	0/0	2989 (19%)	1445	0/0	13.7%	1.7%
↳ pid_0	654	385	0/0	742	391	0/0		
↳ divider_dshift_1	640	351	0/0	742	351	0/0		
↳ divider_dshift_2	672	351	0/0	742	351	0/0		
↳ divider_dshift_3	663	333	0/0	742	351	0/0		
Synth2 (top)	709 (1%)	390	0/0	755 (5%)	391	0/0	6.5%	0.3%
↳ pid_simple_0	708	390	0/0	742	391	0/0		
Synth3 (top)	3326 (5%)	3028	0/0	3526 (23%)	3118	0/0	6.0%	3.0%
↳ mem_ctrl_0	1482	926	0/0	1026	990	0/0		
↳ ac97_0	901	1019	0/0	930	1032	0/0		
↳ ac97_1	901	1019	0/0	930	1032	0/0		
↳ des3_area_0	42	64	0/0	623	64	0/0		
Synth4 (top)	5654 (9%)	5952	0/0	5579 (36%)	6008	0/0	-1.3%	0.9%
↳ des3_perf_0	5654	5952	0/0	5540	6008	0/0		
Synth5 (top)	164 (0%)	224	0/0	165 (1%)	224	0/0	0.6%	0.0%
↳ int_mul_inst	70	32	0/0	71	32	0/0		
↳ display_inst	94	192	0/0	94	192	0/0		
Synth6 (top)	1067 (2%)	1207	2/2	1073 (7%)	1207	2/2	0.6%	0.0%
↳ gng_0	659	797	0/2	697	797	2/2		
↳ uart2spi_0	407	410	0/0	374	410	0/0		
Synth7 (top)	3090 (5%)	1065	1/43	3232 (21%)	1067	1/43	4.6%	0.2%
↳ fixed_pt_sqrt_0	472	18	0/24	460	18	0/24		
↳ graphiti_0	1262	665	0/19	1223	668	1/19		
↳ hight_0	329	134	0/0	502	134	0/0		
↳ lfsr_randgen_0	8	4	0/0	2	4	0/0		
↳ cpu8080_0	1019	243	0/0	1020	243	0/0		
Synth8 (top)	3056 (5%)	2231	2/2	3187 (20%)	2272	2/2	4.3%	1.8%
↳ pci_mini_0	224	290	0/0	263	332	0/0		
↳ pic_0	144	106	0/0	164	107	0/0		
↳ potato_0	2433	1664	2/0	2434	1664	2/0		
↳ pwm_0	159	145	0/0	189	145	0/0		
↳ quadratic_func_0	105	24	0/2	109	24	0/2		
Synth9 (top)	7686 (12%)	2220	0/0	8504 (55%)	2293	0/0	10.6%	3.3%
↳ random_pulse_generator_0	4	33	0/0	4	33	0/0		
↳ sha3_high_throughput_0	7634	2157	0/0	8268	2230	0/0		
↳ simon_core_0	48	27	0/0	47	27	0/0		
Average							**5.1%**	**1.2%**

R/D=RAMB/DSP. For the Regular CAD Flow, LUT percentage represents fraction of entire device. For the Encrypted CAD Flow, it represents fraction of user logic region. *__Note:__ This does not include the fixed cost of the Loader static design.

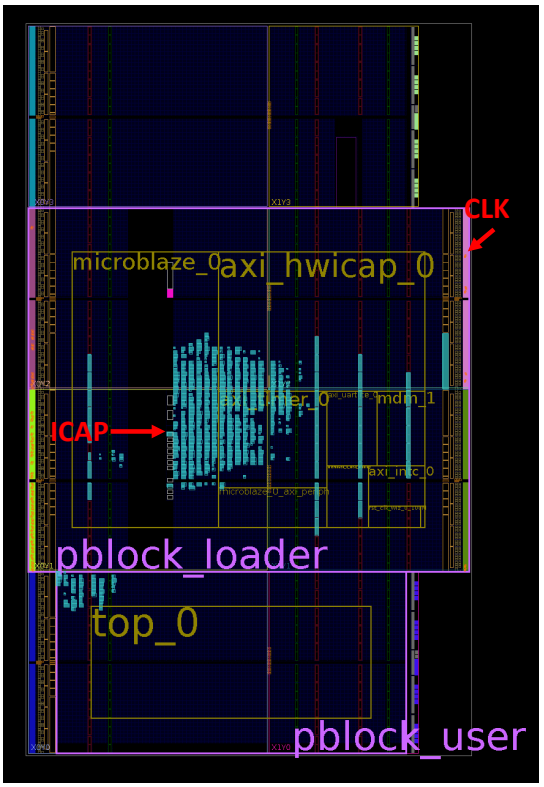


Fig. 8. Partitioning of the FPGA for the Loader Shell and User Design.

Table 2. Resource overhead of our Loader Shell design

Module	LUT	FF	RAMB	DSP
Loader Shell	2052	2701	34	0
↳ MicroBlaze	1321	1159	32	0
↳ AXI HWICAP	575	1367	2	0
↳ AXI Interrupt Controller	68	65	0	0
↳ AXI UartLite	93	110	0	0

regions. Unfortunately, the constraints of this small FPGA and development board are not ideal for our Loader Shell architecture. The *ICAP* primitives, which are used to configure the FPGA, and of which the FPGA only contains two, are both located in the center of the FPGA, requiring our Loader Shell to be placed in the center of the FPGA. In addition, the clock input, which also needs to be located within the static region, is located diagonally across the chip from the ICAP. This results in our Loader Shell needing a large region located in the center of the chip, and the user design being relegated to the bottom of the chip, as shown in Figure 8.

The actual logic utilization of the Loader Shell is quite small, as shown in Table 2, requiring only 2052 LUTs and 2701 FFs. With a larger FPGA, and better board design to place the clock input in a

location near the ICAP, the Loader Shell could be made much smaller, and the user would have access to a much larger portion of the FPGA.

Since the shell/user paradigm is commonly used in cloud FPGA systems, we are confident that a more appropriate partitioning could be achieved in practice.

6.3 CAD Runtime

Table 3 details the CAD runtimes for our various benchmarks.

The *Regular CAD Flow* columns report the synthesis and implementation runtimes for a standard execution of the Vivado tool. No individual runtimes for IP are reported because compilation is performed on the design as a whole.

The *Encrypted IP CAD Flow* columns report the synthesis and implementation runtimes for our proposed flow. The rows include synthesis runtime for both the top-level design, and the encrypted IP, but it is important to recognize that the encrypted IP synthesis is performed by the IP vendor, and not the user. Because the encrypted IP is pre-synthesized and marked as `DONT_TOUCH`, the top-level synthesis time is quite small (our benchmarks contain very little user logic).

The *Encrypt* runtime measures the time to run the encryption Python script discussed in Section 5.1, and ranges from 1.6 seconds to 12.7 seconds.

The *Impl* column represents the Implementation time of the top-level design, which includes the user's top-level design (including encrypted IP), and the Loader Shell. The Loader Shell is pre-implemented, so it has minimal contribution to the runtime.

The final column compares the implementation runtime of the encrypted user design against the standard Vivado flow. We choose to only compare implementation since the synthesis runtimes are not comparable, given most of the synthesis time is spent on the encrypted IP, which is performed by the IP vendor, and not the user.

Most designs see a small reduction in implementation time. We believe this is due to the fact that the encrypted IP flow targets only the user partition, versus the regular flow targeting the entire part. Since the user partition is smaller, there is potentially less exploration performed by the CAD tools (fewer swaps for the placer to consider, and a much smaller routing-resource graph for the router). However, in some designs we did notice a runtime increase, with the largest increase in runtime (+221%) encountered by our largest design. In this case the implementation logs indicated that there was considerable routing congestion when targeting the smaller user partition, which resulted in several more long-running routing iterations.

6.4 Configuration Time

Table 4 details the time to perform configuration, which includes sending the bitstream over UART, processing the bitstream (decryption and patching), and writing the bitstream to the ICAP.

The total configuration time for our proof-of-concept tool is a few minutes, and varies depending on how many frames in the bitstream need to be decrypted and patched. This is orders of magnitude longer than standard configuration which would take tens of milliseconds for this part. However, we have not invested any time into speeding this process up. If configuration time was a priority, there are several improvements that could be made.

First, the majority of the time is spent transmitting over UART. We choose to send the bitstream over UART for simplicity; a faster communication protocol would substantially reduce configuration time.

Second, the remainder of the runtime is mainly attributed to processing performed by the MicroBlaze. The processing time is roughly linear with the number of LUTs to patch, with each LUT that needs decryption and patching contributing about 25–30 milliseconds to the runtime.

Table 3. CAD Runtime of User Flow

Design	Regular CAD Flow		Encrypted IP CAD Flow*			Runtime Difference
	Synth	Impl	Synth	Encrypt	Impl	Impl
Synth1 (top)	104 s	120 s	33 s		119 s	-0.8%
↳ pid	-	-	59 s	7.4 s	-	
↳ divider_dshift	-	-	44 s	-		
Synth1 (top)	104 s	120 s	33 s		119 s	-0.8%
↳ pid	-	-	59 s	7.4 s	-	
↳ divider_dshift	-	-	44 s	7.8 s	-	
Synth2 (top)	58 s	111 s	28 s		108 s	-2.7%
↳ pid_simple	-	-	57 s	8.0 s	-	
Synth3 (top)	76 s	191 s	33 s		191 s	0.0%
↳ mc_top	-	-	59 s	0.0 s	-	
↳ ac97_top	-	-	31 s	8.1 s	-	
↳ des3_area	-	-	47 s	8.5 s	-	
Synth4 (top)	83 s	201 s	39 s		171 s	-14.9%
↳ des3_perf	-	-	78 s	10.3 s	-	
Synth5 (top)	35 s	161 s	29 s		153 s	-5.0%
↳ int_mul_8	-	-	35 s	7.9 s	-	
↳ configurable_display	-	-	34 s	7.1 s	-	
Synth6 (top)	46 s	147 s	31 s		118 s	-19.7%
↳ gng	-	-	39 s	1.6 s	-	
↳ uart2spi	-	-	35 s	1.6 s	-	
Synth7 (top)	87 s	199 s	34 s		238 s	19.6%
↳ fixed_point_sqrt	-	-	45 s	7.4 s	-	
↳ graphiti	-	-	36 s	7.7 s	-	
↳ hight	-	-	42 s	7.6 s	-	
↳ lfsr_randgen	-	-	32 s	6.6 s	-	
↳ cpu8080	-	-	51 s	8.3 s	-	
Synth8 (top)	97 s	194 s	34 s		173 s	-10.8%
↳ pci_mini	-	-	36 s	6.5 s	-	
↳ pic	-	-	43 s	6.9 s	-	
↳ potato	-	-	77 s	8.9 s	-	
↳ pwm	-	-	33 s	6.8 s	-	
↳ quadratic_func	-	-	32 s	7.0 s	-	
Synth9 (top)	619 s	42 s	18 s		135 s	221.4%
↳ random_pulse_generator	-	-	26 s	7.1 s	-	
↳ sha3_high_throughput	-	-	691 s	12.7 s	-	
↳ simon_core	-	-	26 s	7.1 s	-	
Average					+20.8%	

Table 4. Configuration Runtime

Design	ICAP Config Time	UART Send Time	Processing Time	Total
Synth1	111 ms	160 s	99 s	259 s
Synth2	111 ms	160 s	25 s	185 s
Synth3	110 ms	160 s	98 s	258 s
Synth4	110 ms	160 s	166 s	326 s
Synth5	111 ms	160 s	6 s	166 s
Synth6	111 ms	160 s	33 s	194 s

This processing could be significantly sped up by better optimizing the software code, using a faster processor, or by using hardware acceleration of the decryption process.

6.5 Verification Process

Our tool makes several substantial changes to the already complex FPGA CAD process, and given the many possible points of failure, we wanted to be sure that our technique was correctly implementing the design in the FPGA. Our approach to verify correctness of our tools was the following:

(1) Compile the encrypted IP using our proposed vendor flow (Figure 4). Though not shown in previous figures, the vendor flow generates two netlists, one that is encrypted and one that maintains the LUT *INIT* values intact specifically for verification.
(2) Compile the user design using our proposed user flow (Figure 5), this time including the intact netlist. This results in a modified bitstream that still contains the intact LUT *INIT* values and will serve as a “golden" bitstream.
(3) We then modify the Loader software to enable a verification mode. In this mode the Loader compares each decrypted bit with the bit it will patch in the golden bitstream. If they do not match, then the Loader reports an error.
(4) We then perform this verification process on each design by first generating the “golden" modified bitstream, and then using the verification mode to assure no modification to the “golden" bitstream when the loader decrypts and writes the encrypted *INIT* values.

The above process ensures that the process of tracking LUTs through the CAD flow, dealing with CAD tool optimization, and decrypting and patching the bitstream at configuration time, is all working correctly. This method helped us catch and correct many of the subtle optimization issues discussed earlier in the paper.

7 ISSUES AND LIMITATIONS

Despite our approach significantly improving the state-of-the-art for IP protection, there are still a number of limitations and issues that should be identified.

Partial Encryption: One significant limitation of our approach is that the IP is not fully encrypted. The *INIT* values are hidden but the full structure of the netlist is still visible to the user, including the relative connections of LUTs, FFs, adders, and other primitives. While cell names could be removed from the netlist, an attacker could still learn important implementation details of the IP by just looking at the structural layout. In Section 4 we discuss alternative approaches that could provide coarser-grained granularity of encryption thereby hiding more structure of the IP;

however, this would require more significant changes to the CAD tool, and would be more difficult to implement.

However, it should be noted that the current approach to IP encryption in Vivado similarly allows the user to view cell instances and how they are connected, without needing to perform any of the workarounds such as Tcl interrogation or bitstream-to-netlist processing. Thus, at minimum our approach hides the same information as current approaches, but is immune to the attacks that can be performed on current approaches.

Vulnerability to Key Theft: Another limitation is that there is a possibility that the user can potentially steal the encryption keys. This shouldn't be possible, given they are encrypted into the Loader design; however, previous work, such as the Starbleed attack [24], has demonstrated that it is possible to break FPGA encryption technology. Currently this is only the case for certain FPGA families, although there are no guarantees that new vulnerabilities for other FPGA families won't be discovered in the future. In addition, we have not considered the effects of side-channel attacks, such as dynamic power analysis, which could possibly be used to extract the encryption keys from the Loader design. Despite this, our approach is still vastly more effective than current approaches for IP protection, and makes it much harder for an attacker to gain access to protected IP.

In addition, if this approach were used in an environment where the IP customer did not have access to the FPGA or Loader Shell, such as in a cloud environment, then this risk would be removed entirely.

Brute-Force Netlist Recovery Attacks: Despite the protections offered by our proposed technique, there still may be methods to for a determined attacker to recover the LUT *INIT* properties through a brute-force attack. For example, a malicious IP user could alter the proprietary bitstream, select an encrypted LUT and change the metadata to indicate that this LUT is not encrypted. This would likely break the functionality of the design since the LUT would have the incorrect *INIT* property. However, the attacker could then continually regenerate bitstreams with different LUT *INIT* values until the correct value is found.

The feasibility of such an attack would depend on a number of factors. First, the attacker would need a way to validate whether the LUT *INIT* value they are trialing is correct. This may be very difficult, especially since the attacker may not know what feature of the IP this LUT impacts, or how to validate its correctness. Validating the correctness of the LUT *INIT* value would likely require substantial test vectors with perfect test coverage to be sure the LUT is functioning correctly in all possible scenarios.

Second, the number of possible LUT *INIT* values for a k-input LUT is 2^{2^k}, making this attack computationally infeasible for larger LUTs. For example, supposing an attacker is able to generate a new bitstream, configure it to the FPGA via the Loader, and fully validate the functionality of the IP in one minute, it would take four hours to test every possible 3-LUT *INIT* value, 46 days for a 4-LUT, and thousands of years for 5/6-LUTs.

Such an attack could be used to obtain many of the smaller LUT *INIT* values, but would be computationally infeasible for an entire IP, which would naturally contain many larger LUTs.

Simulation and IP Reverse Engineering: A substantial limitation of any encrypted IP approach is that the user is no longer able to perform simulation on the design. Current FPGA tools do not run into this issue as the CAD tool has access to decrypt the IP design for simulation. One way to address this issue would be for the IP vendor to provide a simulation executable that could provide cycle-accurate behavior of the IP, and a plug-in system that would allow this to be integrated into the simulator tool.

Providing a cycle-accurate behavioral oracle would allow a user to try and break the IP encryption by providing various inputs and observing the outputs. It is possible that with machine-learning or

SAT-based attacks a user could gain the full netlist details. However, such an approach would be far more difficult and compute-intensive than the current trivial methods of obtaining encrypted IP netlists described in Section 2.1.

In addition, depending on the implementation of the IP module, this may be computationally infeasible. For example, suppose the IP is an AES encryption module with the encryption key hard-coded into the LUT logic. AES is inherently designed such that even with multiple known plaintext/ciphertext pairs, it is computationally infeasible to determine the encryption key. Thus, even with a cycle-accurate input/output behavioral oracle, it would be impossible to determine the encryption key. However, the simulator executable could still be vulnerable to other attacks, such as memory probing, which could be used to extract the encryption key from the executable.

Feature Restrictions Aside from simulation, there are a number of convenient other design-flow features that would be prohibited when using encrypted IP. Since the user is being provided with a partially encrypted netlist, they do not have access to the original RTL (*soft IP*), and cannot change configuration options that may have otherwise been possible. In addition, debugging features such as Xilinx Integrated Logic Analyzer (ILA) debug probes and device readback would not be possible. Dynamic reconfiguration would also not be possible.

8 CONCLUSION

In this paper we have presented a novel approach for protecting third-party IP on FPGAs. Our approach allows the IP vendor to partially encrypt their IP, and then provide the encrypted IP to the user. The user can then incorporate the encrypted IP into their design, and generate a custom bitstream that can be loaded onto the FPGA. The FPGA contains a special static region that contains the decryption and configuration circuitry, and is able to decrypt the IP during configuration.

We have implemented a proof-of-concept tool that demonstrates our approach, and have evaluated it using a collection of benchmark circuits. Our results show that our approach is able to protect the IP, while only incurring a moderate overhead to resource usage. Our configuration runtime is significantly higher than traditional configuration, but there are many opportunities to speed up the configuration.

We hope that future work will explore other implementations in this vein, allowing FPGA CAD tools to operate on partially encrypted IP, along with the development of new FPGA devices that are designed with this type of IP protection in mind.

REFERENCES

[1] IEEE Computer Society, "IEEE recommended practice for encryption and management of electronic design intellectual property (IP)," IEEE, Dec. 10, 2014, ISBN: 9780738194929.

[2] J. Speith, F. Schweins, M. Ender, M. Fyrbiak, A. May, and C. Paar, "How not to protect your IP – an industry-wide break of IEEE 1735 implementations," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, pp. 1656–1671. arXiv: 2112.04838[cs].

[3] F4PGA. "F4pga/prjxray." (Jun. 23, 2020), [Online]. Available: https://github.com/f4pga/prjxray (visited on 06/23/2020).

[4] F4PGA. "F4pga/prjuray." (Jul. 21, 2020), [Online]. Available: https://github.com/f4pga/prjuray (visited on 07/22/2020).

[5] 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.

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

[7] CHIPS Alliance. "Chipsalliance/f4pga-xc-fasm2bels." (Apr. 10, 2023), [Online]. Available: https://github.com/chipsalliance/f4pga-xc-fasm2bels (visited on 07/24/2023).

[8] J. Yoon, Y. Seo, J. Jang, *et al.*, "A bitstream reverse engineering tool for FPGA hardware trojan detection," in *Conference on Computer and Communications Security (CCS)*, Oct. 15, 2018, pp. 2318–2320.

[9] A. Chhotaray, A. Nahiyan, T. Shrimpton, D. Forte, and M. Tehranipoor, "Standardizing bad cryptographic practice: A teardown of the IEEE standard for protecting electronic-design intellectual property," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, New York, NY, USA: Association for Computing Machinery, Oct. 30, 2017, pp. 1533–1546.

[10] R. McKendrick, K. Faulkner, and J. Goeders, "Assuring netlist-to-bitstream equivalence using physical netlist generation and structural comparison," presented at the International Conference on Field-Programmable Technology (FPT), Dec. 2023.

[11] 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.

[12] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *International Symposium on Hardware-Oriented Security and Trust*, Jun. 2012, pp. 83–88.

[13] P. Subramanyan, N. Tsiskaridze, W. Li, *et al.*, "Reverse engineering digital circuits using structural and functional analyses," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 1, pp. 63–80, Mar. 2014.

[14] W. Li, A. Gascon, P. Subramanyan, *et al.*, "WordRev: Finding word-level structures in a sea of bit-level gates," in *International Symposium on Hardware-Oriented Security and Trust (HOST)*, Jun. 2013, pp. 67–74.

[15] N. Albartus, M. Hoffmann, S. Temme, L. Azriel, and C. Paar, "DANA universal dataflow analysis for gate-level netlist reverse engineering," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 309–336, Aug. 26, 2020.

[16] R. McKendrick, C. Simpson, B. Nelson, and J. Goeders, "Leveraging FPGA primitives to improve word reconstruction during netlist reverse engineering," in *International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2022, pp. 1–5.

[17] T. Kean, "Cryptographic rights management of FPGA intellectual property cores," in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ser. FPGA '02, New York, NY, USA: Association for Computing Machinery, Feb. 24, 2002, pp. 113–118.

[18] L. Gaspar, V. Fischer, T. Güneysu, and Z. C. Jouini, "Two IP protection schemes for multi-FPGA systems," in *International Conference on Reconfigurable Computing and FPGAs*, Dec. 2012, pp. 1–6.

[19] K. Kepa, F. Morgan, K. Kosciuszkiewicz, and T. Surmacz, "SeReCon: A secure reconfiguration controller for self-reconfigurable systems," *International Journal of Critical Computer-Based Systems*, vol. 1, no. 1, pp. 86–103, Feb. 1, 2010.

[20] K. Kepa, F. Morgan, and K. Kosciuszkiewicz, "IP protection in partially reconfigurable FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2009, pp. 403–409.

[21] 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.

[22] G. Provelengios, D. Holcomb, and R. Tessier, "Power wasting circuits for cloud FPGA attacks," in *International Conference on Field-Programmable Logic and Applications (FPL)*, Aug. 2020, pp. 231–235.

[23] 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.

[24] 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*, Aug. 2020, pp. 1803–1819.