US 20190305927A1

(19) **United States**

(12) **Patent Application Publication**

Bhunia et al.

(10) **Pub. No.: US 2019/0305927 A1**

(43) **Pub. Date: Oct. 3, 2019**

(54) **BITSTREAM SECURITY BASED ON NODE LOCKING**

(71) Applicant: **University of Florida Research Foundation Incorporated**, Gainesville, FL (US)

(72) Inventors: **Swarup Bhunia**, Gainesville, FL (US); **Robert A. Karam**, Gainesville, FL (US); **Tamzidul Hoque**, Gainesville, FL (US)

(21) Appl. No.: **16/081,027**

(22) PCT Filed: **Mar. 17, 2017**

(86) PCT No.: **PCT/US2017/023017**

§ 371 (c)(1),
(2) Date: **Aug. 29, 2018**

Related U.S. Application Data

(60) Provisional application No. 62/310,543, filed on Mar. 18, 2016.

Publication Classification

(51) **Int. Cl.**

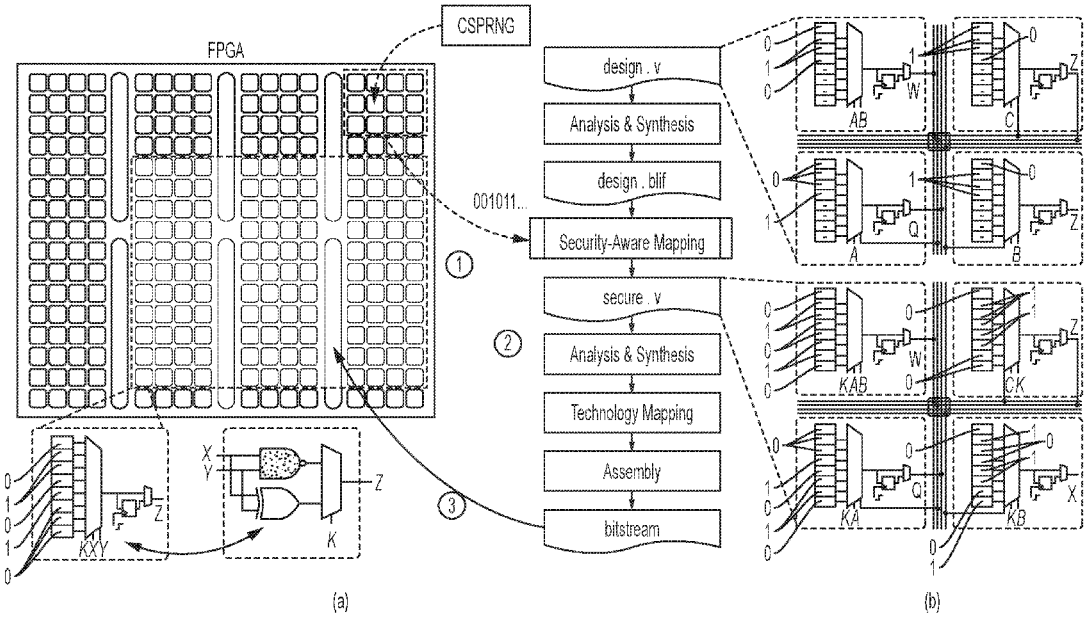
H04L 9/00	(2006.01)
H03K 19/177	(2006.01)
G06F 21/44	(2006.01)
G06F 21/76	(2006.01)
H04L 29/06	(2006.01)
H04L 9/08	(2006.01)

(52) **U.S. Cl.**

CPC ***H04L 9/002*** (2013.01); ***H03K 19/17768*** (2013.01); ***G06F 21/44*** (2013.01); *H04L 2209/16* (2013.01); ***H04L 63/0457*** (2013.01); ***H04L 9/0866*** (2013.01); ***G06F 21/76*** (2013.01)

(57) **ABSTRACT**

A technique to generate node locked bitstreams for FPGAs to simultaneously protect against malicious reconfiguration as well as FPGA IP piracy is provided. According to some aspects, modifications in FPGA architecture along with an associated mapping flow enable authenticating and programming a device in a way that maintains FPGA security while requiring low overhead. The technique is more robust against side channel and destructive reverse-engineering attacks in comparison with key-based encryption methods, and has less area, power, and latency overhead. The node locked bitstream approach is attractive in many existing and emerging applications including IoTs, which may require field upgrade of FPGA.



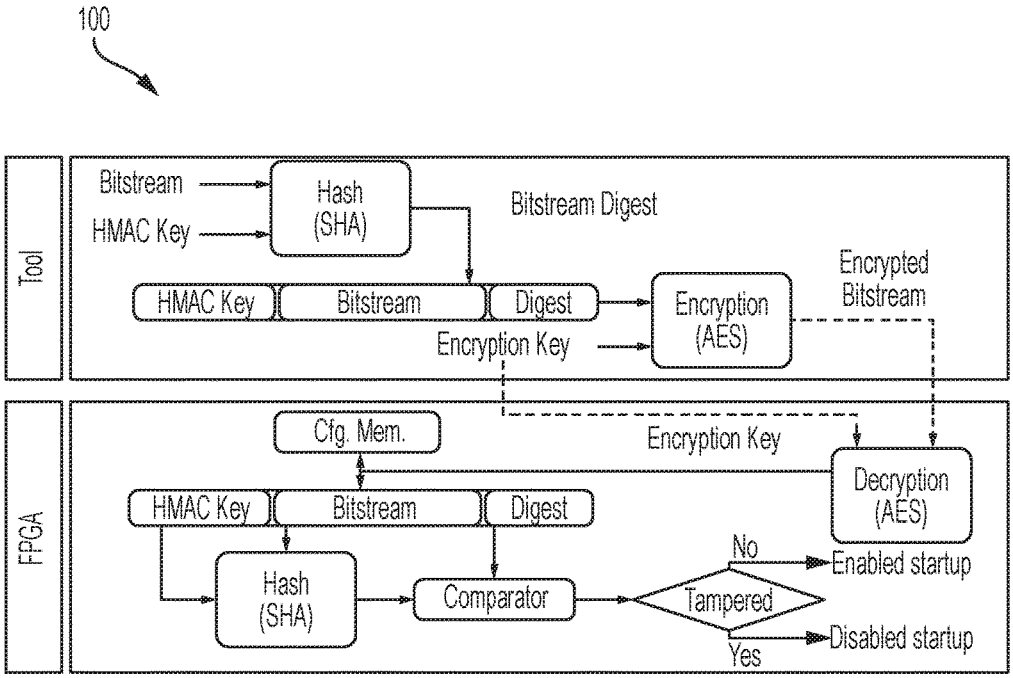

100
Tool
Bitstream
HMAC Key
Hash (SHA)
Bitstream Digest
HMAC Key
Bitstream
Digest
Encryption (AES)
Encrypted Bitstream
Encryption Key
FPGA
Cfg. Mem.
Encryption Key
Decryption (AES)
HMAC Key
Bitstream
Digest
Hash (SHA)
Comparator
Tampered
No
Enabled startup
Yes
Disabled startup

FIG. 1

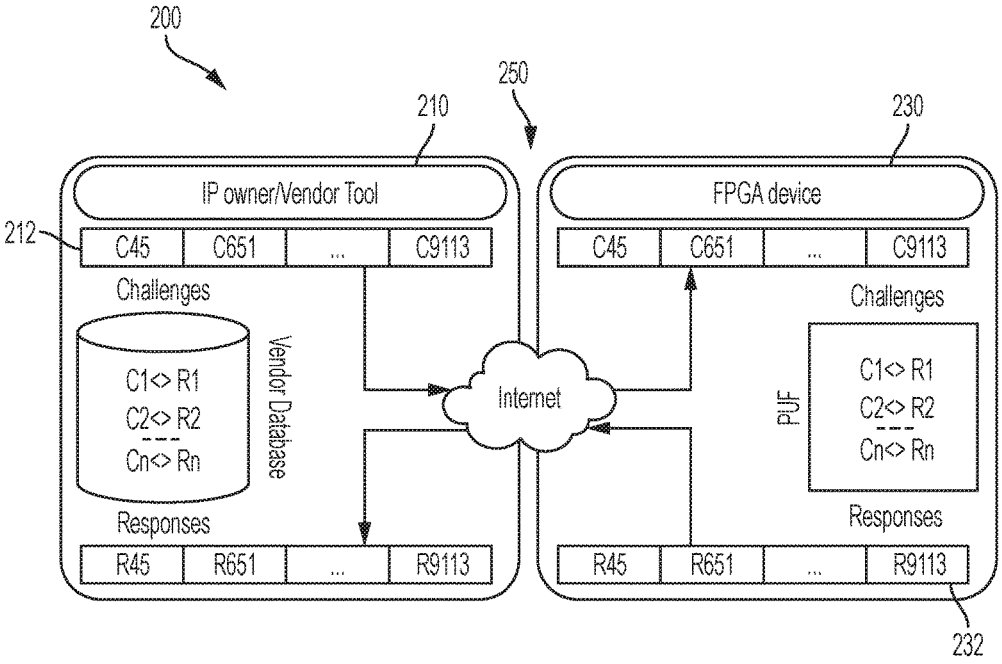

200
250
210
230
IP owner/Vendor Tool
FPGA device
212
C45
C651
...
C9113
Challenges
C1<> R1
C2<> R2
Cn<> Rn
Vendor Database
Internet
PUF
Responses
R45
R651
R9113
232

FIG. 2

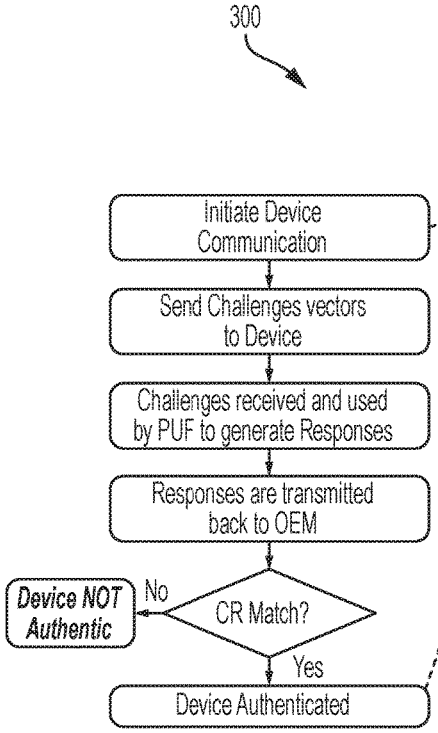

300
Initiate Device Communication
Send Challenges vectors to Device
Challenges received and used by PUF to generate Responses
Responses are transmitted back to OEM
CR Match?
No
Device NOT Authentic
Yes
Device Authenticated
(a)

FIG. 3a

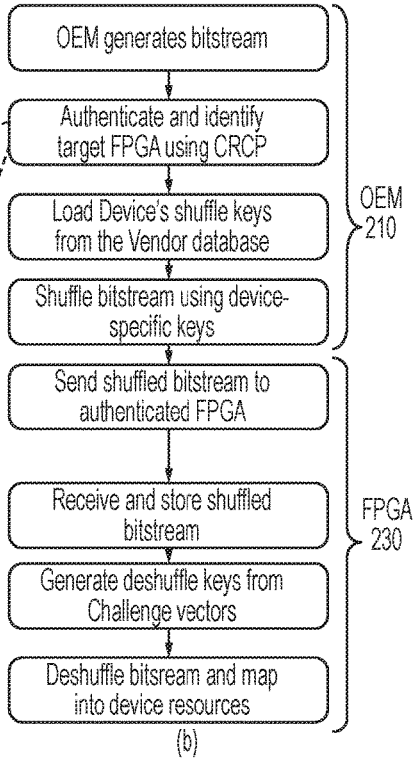

OEM generates bitstream
Authenticate and identify target FPGA using CRCP
Load Device's shuffle keys from the Vendor database
Shuffle bitstream using device-specific keys
OEM 210
Send shuffled bitstream to authenticated FPGA
Receive and store shuffled bitstream
Generate deshuffle keys from Challenge vectors
Deshuffle bitsream and map into device resources
FPGA 230
(b)

FIG. 3b

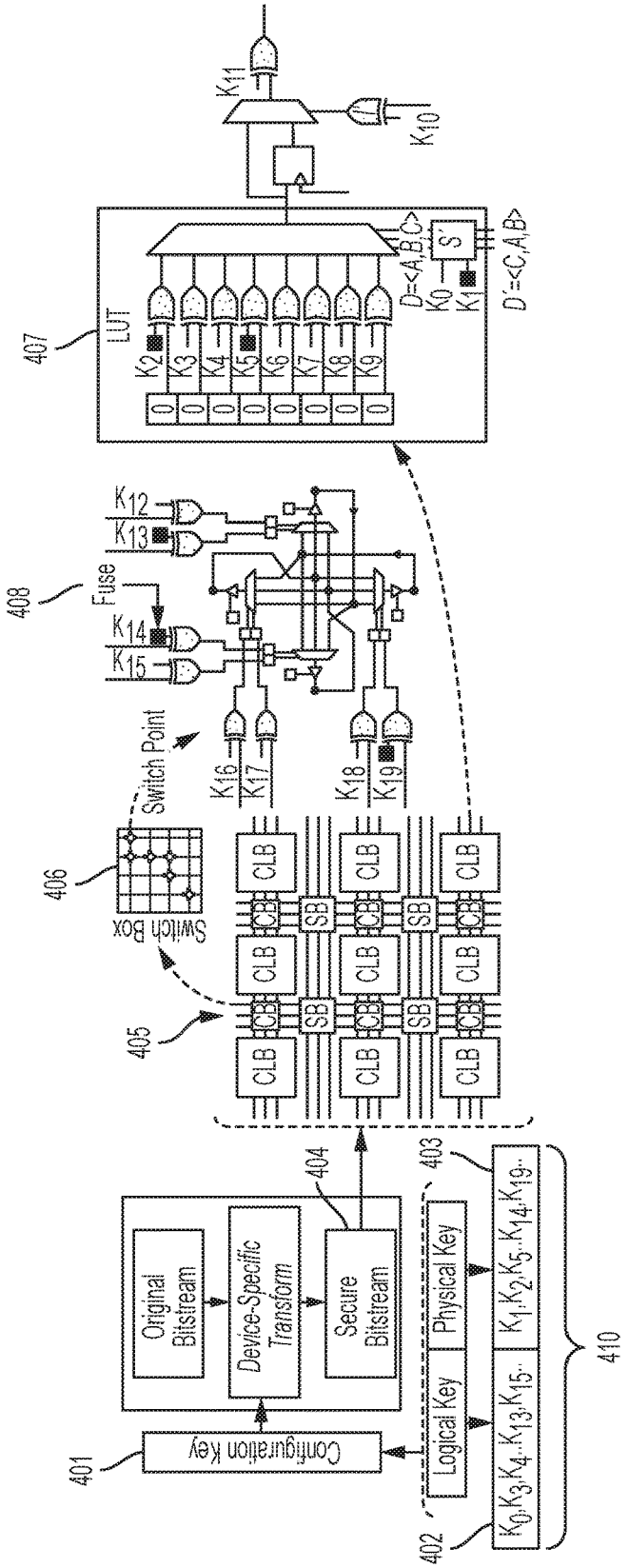

Configuration Key
Original Bitstream
Device-Specific Transform
Secure Bitstream
Logical Key
Physical Key
$K_0,K_3,K_4..K_{13},K_{15}..$
$K_1,K_2,K_5..K_{14},K_{19}..$
401
402
403
404
405
406
407
408
410
LUT
Fuse
Switch Box
Switch Point
CLB
SB
CB
S'
D=<A,B,C>
D=<C,A,B>

FIG. 4

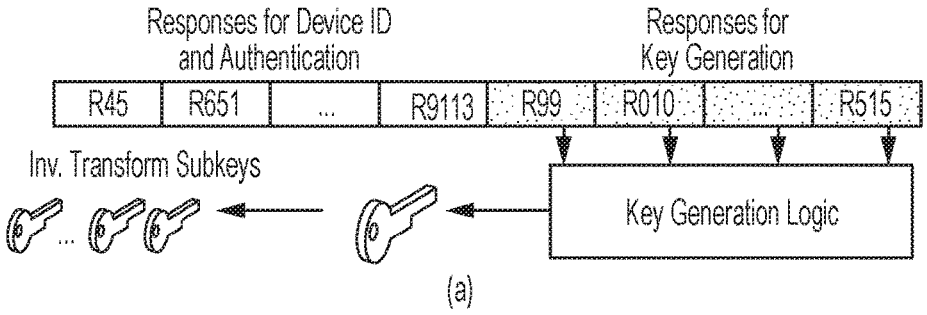

Responses for Device ID and Authentication
Responses for Key Generation
R45
R651
...
R9113
R99
R010
R515
Inv. Transform Subkeys
Key Generation Logic
(a)

FIG. 5a

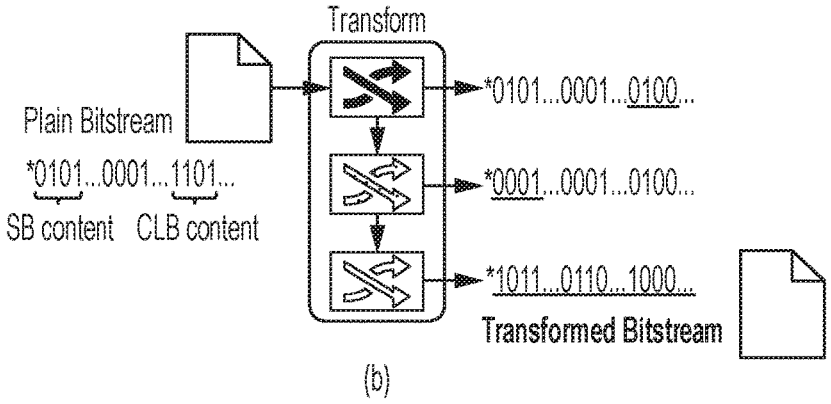

Transform
Plain Bitstream
*0101...0001...1101...
SB content
CLB content
*0101...0001...0100...
*0001...0001...0100...
*1011...0110...1000...
Transformed Bitstream
(b)

FIG. 5b

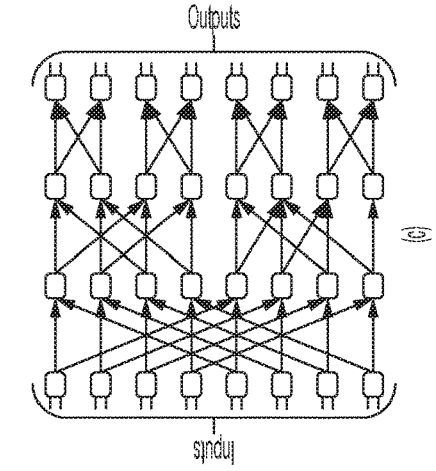


FIG. 6c

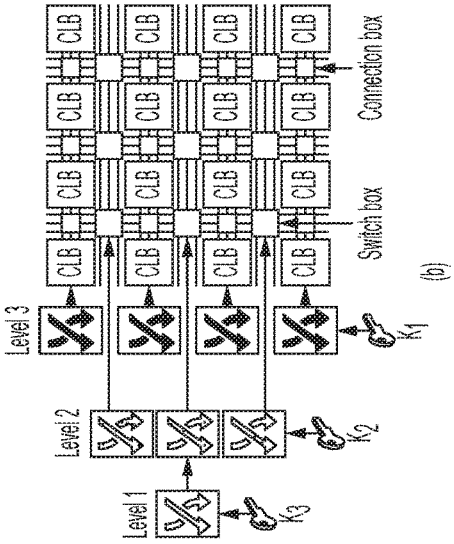


FIG. 6b

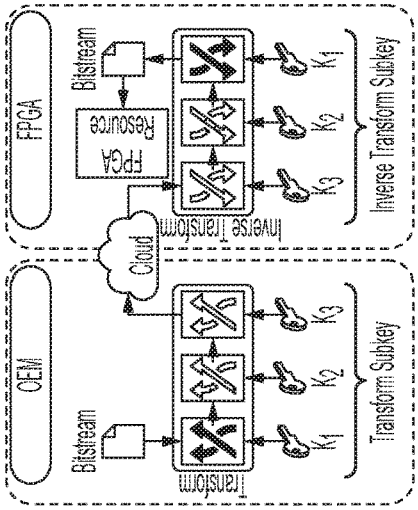


FIG. 6a

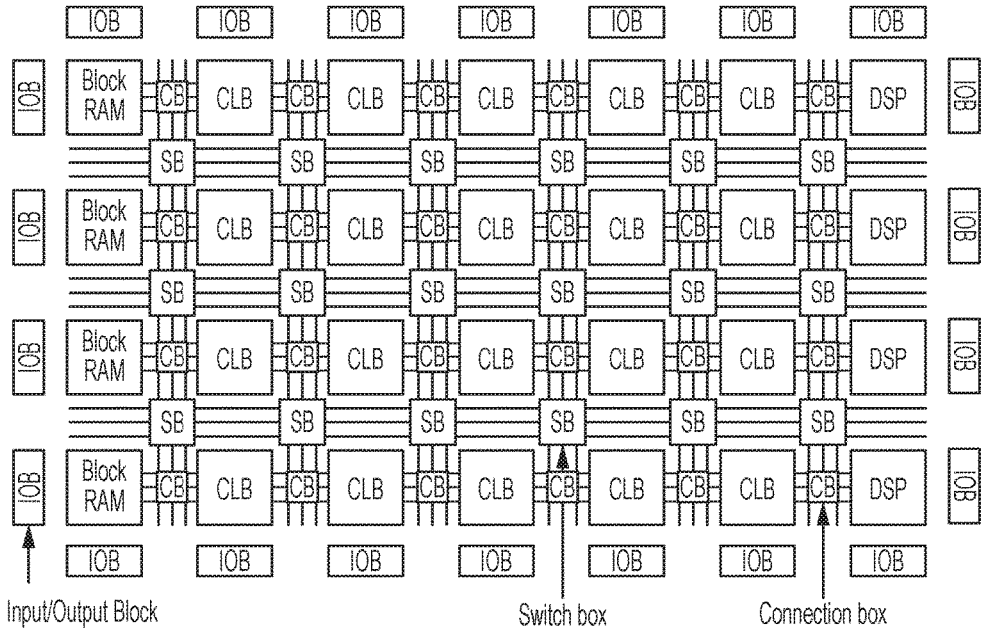

IOB
Block RAM
CB
CLB
SB
DSP
Input/Output Block
Switch box
Connection box

FIG. 7

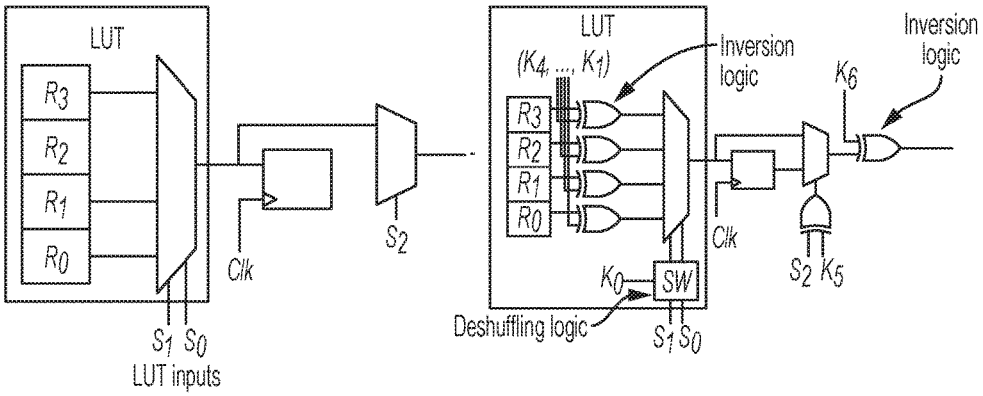


FIG. 8

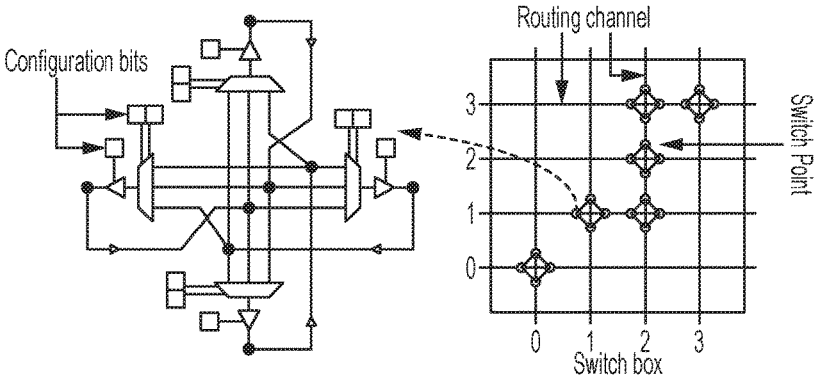


FIG. 9

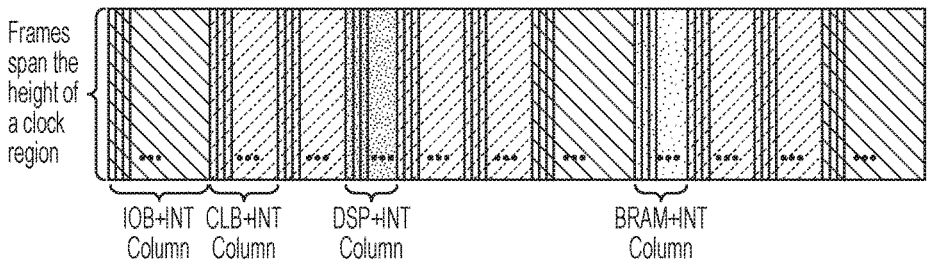


FIG 10

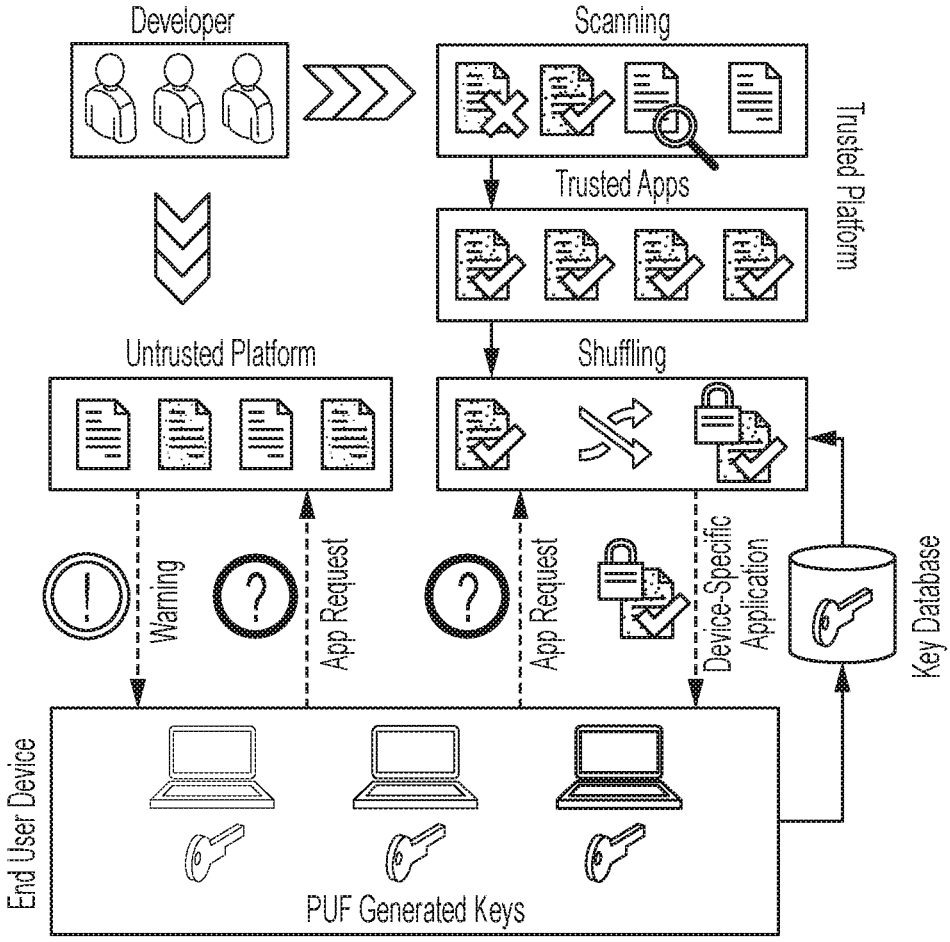

Developer
Scanning
Trusted Platform
Trusted Apps
Untrusted Platform
Shuffling
Warning
App Request
App Request
Device-Specific Application
Key Database
End User Device
PUF Generated Keys

FIG. 11

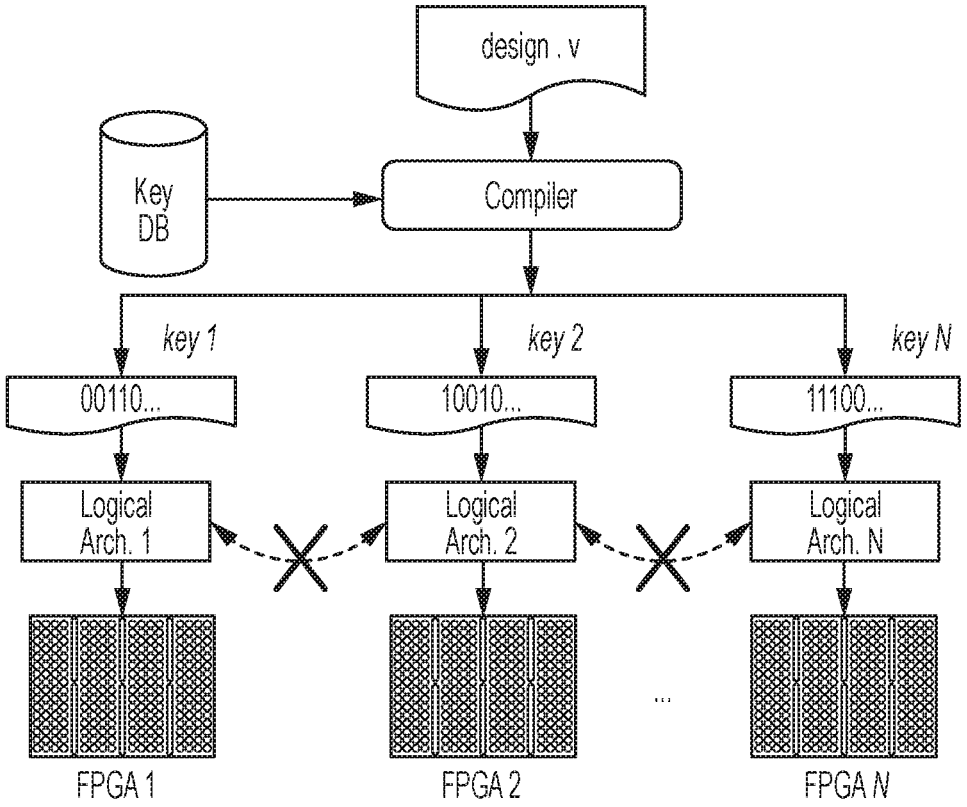

design . v
Key DB
Compiler
key 1
key 2
key N
00110...
10010...
11100...
Logical Arch. 1
Logical Arch. 2
Logical Arch. N
...
FPGA 1
FPGA 2
FPGA N

FIG. 12

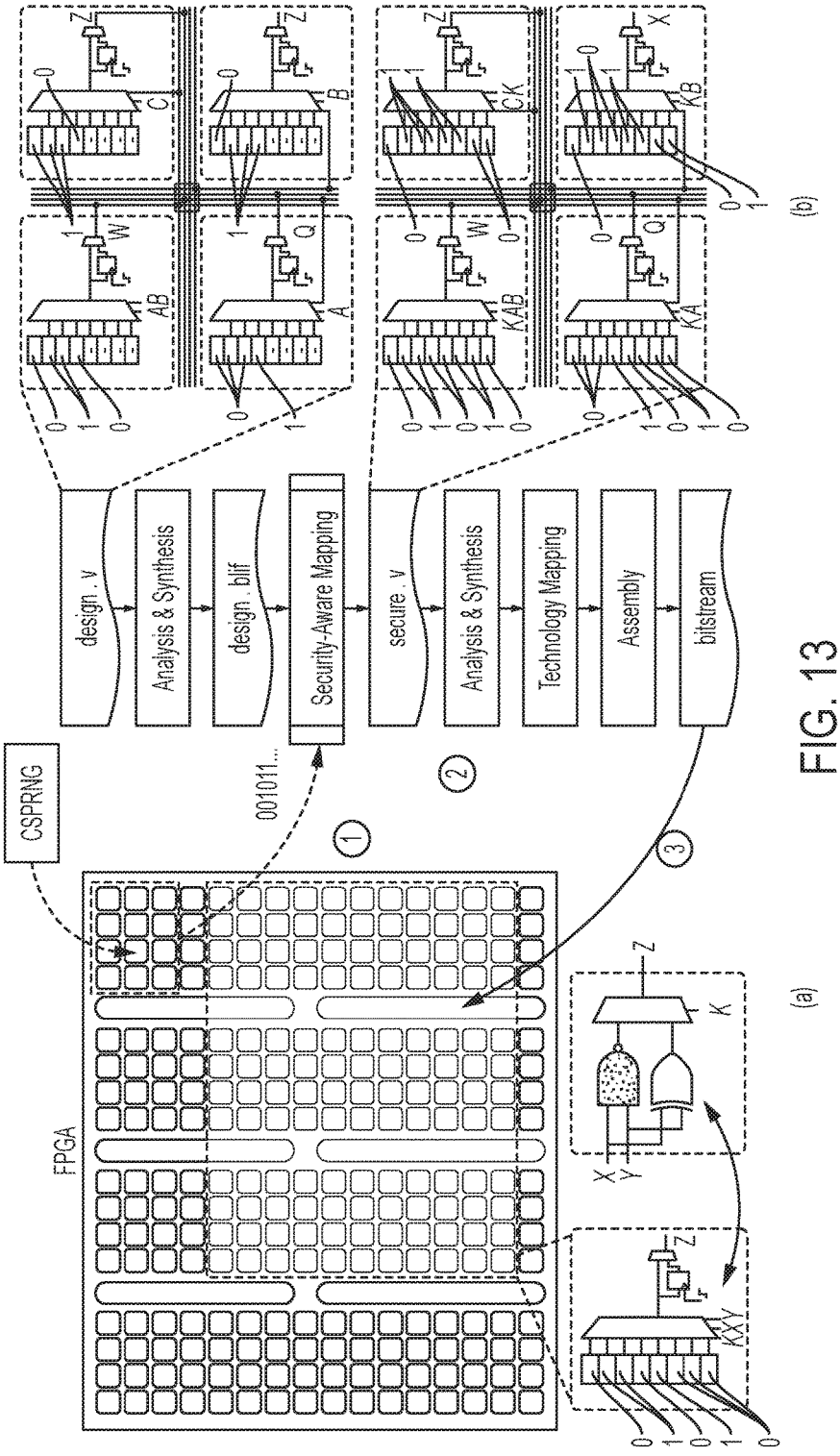

design . v
Analysis & Synthesis
design . blif
Security-Aware Mapping
secure . v
Analysis & Synthesis
Technology Mapping
Assembly
bitstream
CSPRNG
001011...
FPGA
(a)
(b)

FIG. 13

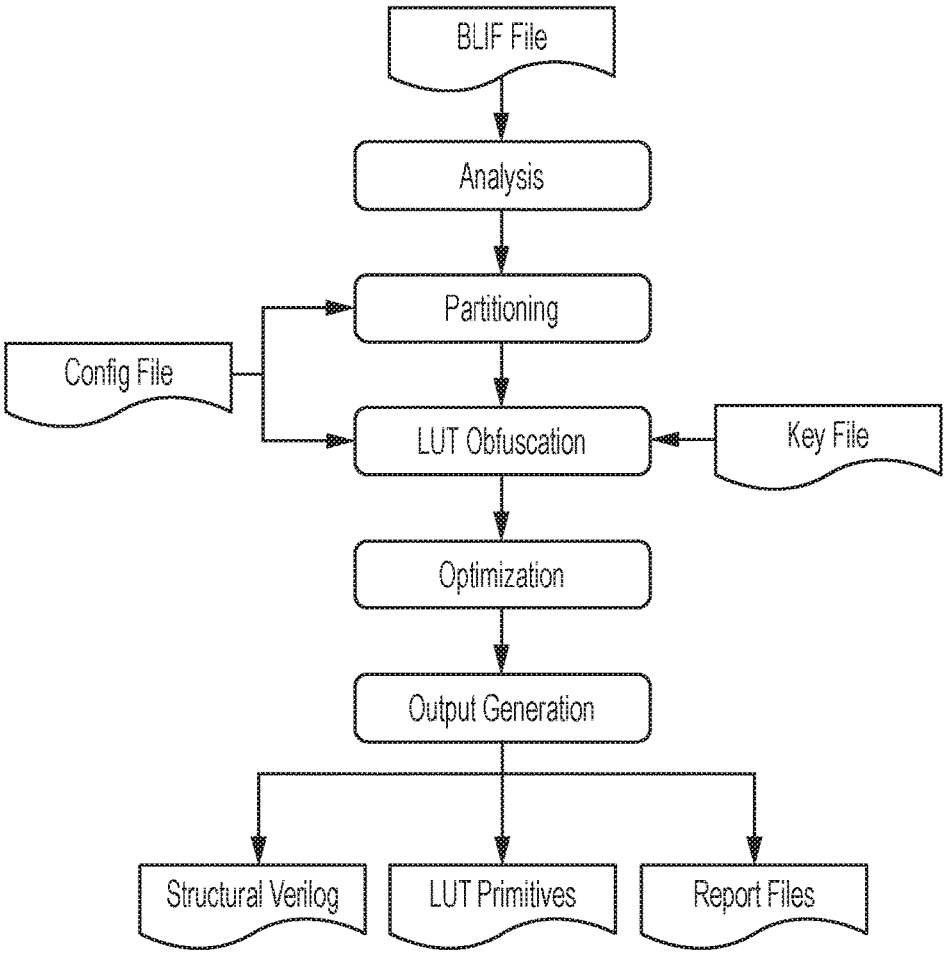

BLIF File
Analysis
Partitioning
Config File
LUT Obfuscation
Key File
Optimization
Output Generation
Structural Verilog
LUT Primitives
Report Files

FIG. 14

BITSTREAM SECURITY BASED ON NODE LOCKING

RELATED APPLICATIONS

[0001] This application claims priority to and the benefit of U.S. Provisional Patent Application No. 62/310,543, entitled "BITSTREAM SECURITY BASED ON NODE LOCKING," filed Mar. 18, 2016. The entire contents of the foregoing are hereby incorporated herein by reference.

BACKGROUND OF INVENTION

[0002] Embedded and wearable computing devices have proliferated in recent years in a large diversity of form factors, performing cooperative computation to provide the new regime of Internet-of-Things (IoT). This proliferation trend is expected to continue, with an estimated 50 billion smart, connected devices by 2020. A key feature in such devices is the need for in-field reconfigurability to adapt to changing requirements in energy-efficiency, functionality, and security. Field Programmable Gate Arrays (FPGAs) have emerged as a popular architecture for addressing this reconfigurability demand. FPGAs provide a high flexibility compared to custom Application-Specific Integrated Circuit (ASIC), while consuming less energy than designs based on firmware running in microcontrollers. Furthermore, FPGA-based designs are known to be more secure than both ASIC and microcontrollers against supply-chain attacks, e.g., design details are not exposed to foundries or entrusted outsourcing.

[0003] Bitstreams contain configuration information for programming a programmable device, such as an FPGA. FPGA bitstreams are susceptible to a variety of attacks, including unauthorized reprogramming, reverse-engineering, and cloning/piracy. Therefore there is a need to provide protection of FPGA bitstreams, both during wireless reconfiguration and after in-field deployment in FPGA-based designs.

BRIEF SUMMARY

[0004] Disclosed herein is an approach to FPGA security that provides protection against in-field bitstream reprogramming as well as Intellectual Property (IP) piracy, while permitting wireless reconfiguration without encryption.

[0005] The inventors have recognized and appreciated that traditional countermeasures against FPGA bitstream attacks, such as shielding, noise injection, etc., use more energy than desired for most modern embedded and IoT devices that have aggressive energy constraints. The present disclosure details aspects of an approach to FPGA security, which can prevent unauthorized in-field reprogramming as well as FPGA IP piracy without encryption. In some embodiments, a node-locked bitstream approach, where the device-to-bitstream association is changed from device to device, is employed.

[0006] According to some embodiments, a programmable device is provided. The programmable device may include an external interface, a first circuit configured to generate an identifier and a second circuit configured to transmit through the external interface at least one response to one or more messages received through the external interface. At least a portion of the at least one response may be based at least in part on the identifier. The programmable device may further include a third circuit configured to perform a de-obfuscating function on a bitstream. The de-obfuscating function may be based at least in part on the identifier. According to some embodiments, the programmable device may be a field programmable gate array (FPGA). The at least a portion of the identifier generated by the first circuit may be based on a plurality of selectively blown fuses in the programmable device. At least a portion of the identifier may have a value that varies over time. The third circuit may include at least one sub-circuit configured to selectively permutate the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier. The third circuit may include a plurality of sub-circuits, connected in series, wherein each of the plurality of sub-circuits is configured to selectively permutate the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier.

[0007] According to some embodiments, a method of securely programming a programmable device is provided. The method may include obtaining an identifier from the programmable device; obfuscating a bitstream based at least in part on the identifier; and sending the obfuscated bitstream to the programmable device. Obtaining the identifier may include sending a sequence of challenges to the programmable device; receiving a sequence of responses to the sequence of challenges from the programmable device; and determining, based on the sequence of responses, the identifier for the programmable device. The method of securely programming a programmable device may further include authenticating the programmable device based on the identifier in relation with an authorized identifier list. Authenticating the programmable device based on the identifier in relation with an authorized identifier list may include obtaining the authorized identifier list from an external source. Obtaining the authorized identifier list from an external source may include communicating with the external source using secure communications. Obfuscating the bitstream may include permutating the bitstream. Obfuscating the bitstream may also include iteratively permutating the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier. Obfuscating the bitstream further may include generating a key based on the identifier and obfuscating the bitstream by performing a plurality of obfuscation functions. Each of the plurality of obfuscation functions may be based on the key. Performing a plurality of obfuscation functions may include iteratively permutating the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the key. Obfuscating the bitstream based on the at least one identifier may include applying a plurality of permutation levels. The plurality of permutation levels may have a first level, a second level and a third level. The first level may include permutation of portions of the bitstream that specify an input ordering of a look up table (LUT); the second level may include permutation of the portion of the bitstream that specifies a content of the LUT and the third level may include a block based permutation of the entire bitstream.

[0008] According to some embodiments, a method of securely operating a programmable device that receives a programming bitstream is provided. The method may include generating a pseudo-random identifier and transmitting a sequence of responses based on the identifier in response to receiving a sequence of challenges. At least a

portion of the sequence of responses may be based at least in part on the identifier. The method may also include deobfuscating a received bitstream based on the identifier; and programming programmable circuitry within the programmable device based on the de-obfuscated bitstream. De-obfuscating the bitstream based on the identifier may include permutating the bitstream based on the identifier. De-obfuscating the bitstream based on the identifier may include transforming the bitstream based on a plurality of fuses in the programmable device that are selectively blown. De-obfuscating the bitstream based on the identifier may further include applying a plurality of permutation levels. The plurality of permutation levels further may include a first de-obfuscation level, a second de-obfuscation level and a third de-obfuscation level. The first de-obfuscation level may include permutating the bitstream on a first portion of the programmable device; the second de-obfuscation level may include permutating the bitstream on a second portion of the programmable device; the third de-obfuscation level may include permutating the bitstream on a third portion of the programmable device.

[0009] The foregoing is a non-limiting summary of the invention, which is defined by the appended claims.

BRIEF DESCRIPTION OF DRAWINGS

[0010] Various aspects and embodiments will be described with reference to the following figures. It should be appreciated that the figures are not necessarily drawn to scale. In the drawings, each identical or nearly identical component that is illustrated in various figures may be represented by a like numeral. For purposes of clarity, not every component may be labeled in every drawing.

[0011] FIG. **1** is a schematic diagram for an exemplary flow for FPGA bitstream encryption and authentication;

[0012] FIG. **2** is a schematic diagram for an exemplary Challenge/Response-based Communication Protocol (CRCP) in some embodiments;

[0013] FIG. **3***a* is a schematic diagram showing an exemplary system flow when the Challenge/Response Communication Protocol (CRCP) identifies and authenticates a device in some embodiments;

[0014] FIG. **3***b* is a schematic diagram showing an exemplary system flow of the node locked bitstream approach in some embodiments;

[0015] FIG. **4** is a schematic diagram of an exemplary mapping flow in some embodiments;

[0016] FIG. **5***a* is a schematic diagram showing an exemplary bitstream transform key generation process, according to some embodiments;

[0017] FIG. **5***b* is a schematic diagram for an exemplary three level transformation scheme;

[0018] FIG. **6***a* is a schematic diagram for an exemplary three level transformation scheme showing three levels of transformation by the Vendor tool and three levels of inverse-transformation in the FPGA;

[0019] FIG. **6***b* is a schematic diagram showing an exemplary inverse transformation in some embodiments;

[0020] FIG. **6***c* is a schematic diagram for an example Level 1 inverse transform network operating on 16 bits of input, using 4 bits of key to transform data;

[0021] FIG. **7** is a schematic diagram showing a simplified exemplary architecture of an FPGA fabric containing CLBs, Block RAMs, DSP blocks, routing resources, and IO Blocks in some embodiments;

[0022] FIG. **8** is a schematic diagram of an example LUT structure containing SRAM cell and MUX with peripheral logics such as Flip Flops and MUX according to one embodiment. Various inversion and transformation logic is applied to implement permutation and selective inversion based security;

[0023] FIG. **9** is a schematic diagram showing an example of routing resources such as a switch box and gate level design of switch points;

[0024] FIG. **10** is a schematic diagram showing an exemplary structure of a bitstream frame containing bits for JOB, CLB, BRAM, DSP, and their interconnects according to prior art [Ref. 19]. A single frame may represent a tiny portion of the physical FPGA layout. The whole design may be implemented through a large number of such frames;

[0025] FIG. **11** is a schematic diagram of an exemplary protocol for PUF-based application security using a trusted cloud server;

[0026] FIG. **12** is a schematic diagram showing an exemplary scheme of key-based bitstream obfuscation;

[0027] FIG. **13** is a schematic diagram showing an exemplary security-aware mapping for FPGA bitstreams;

[0028] FIG. **14** is a schematic flow diagram of an exemplary software flow leveraging FPGA dark silicon for design security through key-based obfuscation.

DETAILED DESCRIPTION OF INVENTION

[0029] The inventors have recognized and appreciated security techniques for programmable devices that ameliorate limitations of existing security techniques, improving the usefulness of programmable devices for low cost, widely used devices, such as those that can be used to implement the IoT. For example, on-board encryption technologies used in modern FPGA-based devices incur large area and power overhead, particularly for area/energy-constrained applications. Furthermore, since the attacker typically has physical access to the device, most on-board encryption techniques are susceptible to side-channel attacks, e.g., by key extraction through power profile signatures [Ref. 1]. Moreover, they are still vulnerable to piracy and malicious alteration during in-field upgrade.

[0030] Therefore, there exists a need for a secure programmable device and programming method to safeguard against bitstream attacks, without incurring large area and energy overhead. Techniques that provide one or more of these characteristics are described herein. The inventors have recognized that two primary attack models exist for programmable devices: unauthorized reprogramming and reverse engineering. Unauthorized reprogramming using a bitstream maliciously modified by insertion of a Trojan may alter system functionality, leak information, or cause a failure. A reverse-engineered design can be sold as original, leading to Intellectual Property (IP) piracy.

[0031] To combat unauthorized reprogramming in the first attack model, the inventors have recognized that bitstream encryption may be used. FIG. **1** shows an example of such an encryption process **100**. Bitstream encryption using a symmetric cypher such as Triple DES (3DES) or AES, is typically used for protecting the configuration files in the bitstream. An decryption engine inside the FPGA is used to decrypt the configuration bits before it is mapped to FPGA resources. In many cases, these keys are generated by a vendor's mapping tool and are transmitted along with the

bitstream itself. If transmitted over a network, this can greatly compromise system security.

[0032] The use of FPGA-specific keys has also been investigated. For example, a public key cryptography scheme which uses a trusted third party for key transportation and installation has been proposed [Ref. 2]. However, this scheme relies on the assumption that the FPGA has built-in fault tolerance and tamper resistance countermeasures, including multiple instances of identical cryptographic blocks for detecting operational faults, which would not be viable for area- and power-limited systems.

[0033] FPGAs like the Xilinx Zynq-7000 [Ref. 3] integrate an SoC and FPGA in a single system, and use public key cryptography for authentication during a secure boot process. The public key used to decrypt configuration files is stored in the device's nonvolatile memory, and its integrity is checked before every use [Ref. 4]. These security measures rely on a CPU to control the secure boot process, and are therefore viable only in such hybrid systems. A common feature among these encryption-based techniques is that key storage is resilient to physical attacks; however, this feature is often lacking in practice [Ref. 5].

[0034] Mathematically, the encryption algorithms are known to be highly secure against brute force attacks. However, successful Side-Channel Attacks (SCA) have been mounted against these systems, enabling decryption of the IP [Refs. 6-8]. The inventors have recognized that unless additional countermeasures are in place (e.g. obfuscation), an adversary can easily convert the bitstream to a netlist [Ref. 9], making malicious modifications possible. Therefore, even state-of-the-art methods for FPGA bitstream encryption cannot ensure IP security.

[0035] On the other hand, to counter the second model of bitstream attack such as bitstream tampering, hashed codes are often used as authentication, similar to checksums on software. While this can help prevent malicious modification, it cannot prevent reverse engineering of the IP. This method also provides key storage in nonvolatile memory, for which successful differential power analysis (DPA) attacks have been demonstrated [Ref. 10].

[0036] As discussed above, the inventors have recognized that neither encryption nor authentication alone is capable of protecting bitstreams against a motivated attacker. To mitigate this, it is desirable to design an IP protection scheme that has the following properties:

[0037] Resilient to brute force, side channel, and destructive reverse engineering attacks;

[0038] Independent of non-volatile storage, which is known to be vulnerable;

[0039] Economical in terms of production and recurring costs;

[0040] Low area and power overhead, and viable for use in IoT and other embedded devices;

[0041] Capable of restricting reconfiguration to authorized parties.

[0042] The inventors have appreciated and recognized the need to provide bitstream security against both primary bitstream attack modes. An aspect of the present disclosure provides a device and method based on changing the underlying architectural configuration of FPGA from device to device such that a bitstream can only work in a specific FPGA device. In some embodiments, an application mapping tool, such as may be used in initially programming or reprogramming an FPGA, queries a device to learn about its architecture and then generates an appropriate node-locked bitstream (NLB) for a specific device. The query may be clone using a Challenge/Response (CR) device authentication approach. The tool then uses device-specific keys to generate a bitstream. To be effective, the NLB is unique to each device according to aspects of an embodiment. In other words, a bitstream compiled for one device may not physically map the same functions on a second. Furthermore, in some embodiments architectural changes may be achieved post-silicon, making the device and method compatible with existing processes while requesting minor adjustments to software tool flow. In some embodiments, device authentication does not rely on a key stored in a nonvolatile memory (NVM). Rather, in some embodiments, a device may use a pseudo-random function to generate an identifier for itself that may be time varying, but revealed in the CR protocol.

[0043] Example embodiments of such a programmable device with protocols for device identification, authentication, reconfiguration and secure transmission of bitstreams to remote devices during field upgrade are discussed in detail below.

[0044] Furthermore, details of a security analysis are provided below demonstrating protection in some embodiments against key extraction from a bitstream and bitstream reverse-engineering with significantly decreased area and power overhead compared with area-optimized encryption blocks.

[0045] The inventors have recognized that for devices that support in-field upgrades, preventing unauthorized reprogramming of a device and ensuring unauthorized or counterfeit devices do not receive valuable upgrades are important security goals, and additional steps may be taken instead of or in addition to a Challenge Response Communication Protocol (CRCP). In one embodiment, through the use of Challenge/Response (CR)-based device authentication and device-specific keys for IP antipiracy, a solution may be provided to render FPGAs more secure against IP piracy and unauthorized reprogramming. According to an aspect, the authentication protocol involves communication between the FPGA Vendor and the Original Equipment Manufacturer (OEM), which produces the bitstream.

[0046] In one non-limiting example, CRCP is an authentication mechanism transmitting through an external interface a sequence of 64 bit Challenges as inputs to a circuit such as a Physically Unclonable Function (PUF) on the FPGA. In some embodiments, the circuit may be a MECCA PUF. Although 64 bit Challenges are used as input, any other suitable bit length may be used as the sequence of Challenges to increase the difficulty for brute force attacks to deduce the sequence. A circuit on the FPGA may be used to generate a sequence of Responses to the sequence of Challenges. The sequence of Responses is unique to the particular device and in some embodiments may be based on a unique identifier to the particular device. The unique identifier may include physical modifications performed by the FPGA manufacturer; the identifier may also include time-variant modifications based on a logical-key as described in further detail in the sections below.

[0047] FIG. **2** shows an illustrative example of the CRCP-based authentication process **200**, while FIGS. **3***a* and **3***b* show another exemplary CRCP-based authentication process **300**. To authenticate a device, the OEM **210** sends a predetermined number of challenges **212** through an external interface **250**, and the device **230** responds in turn, as

shown in the illustrative examples in FIG. **2** and FIG. **3** by transmitting a sequence of responses **232** through the external interface. In some embodiments, the number of challenges may be variable over time. CR pairs may be batched and sent to the Vendor server, which returns a set of device-specific identifiers. In some embodiments, the Vendor/OEM communication may be through secure channels, for example via encrypted communication using industry standard methods. According to one aspect, the authentication scheme may comprise two important components: 1) the Vendor precharacterizes the devices after fabrication through an enrollment process, which ensures that only legitimate devices will receive in-field upgrades; 2) the software tools used by the OEM have access to the Vendor database containing an authorized identifier list.

[0048] In some embodiments, once the device has been authenticated, an upgrade procedure using a bitstream may begin. Because the bitstream may be wirelessly transmitted to the device and stored in NVM, it is important to transform it in some way to prevent reverse engineering. According to an aspect of some embodiments, Node Locking a bitstream is provided to an individual FPGA using a two-layer obfuscation scheme which uses both physical and logical key-based architectural modifications to provide a unique identifier to ensure a unique bitstream-to-device mapping. Example techniques to implement the two-layer obfuscation scheme are provided herein.

[0049] According to an aspect, the first of two obfuscation layers is based on physical architectural modifications to the underlying FPGA fabric. This layer is comprised of a network of fuses programmed by the FPGA manufacturer after fabrication. The selectively blown fuses may represent a portion of the unique identifier to the FPGA device as manufactured in order to enable bitstream node-locking. In some embodiments, the programming of the network of fuses may be pseudo-random. Devices which do not need reprogramming during their lifetimes (e.g. a printer) may use only the physical obfuscation layer and retain a high degree of security through architectural diversity. Furthermore, in some embodiments because each FPGA is programmed with its vendor's specific toolset, the physical modification may prevent the fabrication facility from overproducing and selling functional devices.

[0050] In some embodiments, once the device has been authenticated, the bitstream may be modified by the vendor tool prior to FPGA programming. Based on the configuration of the physical modifications, LUT content bits, programmable interconnect switches, or other configuration bits may be inverted, permuted, or otherwise transformed to fit the target architecture. In some embodiments, no additional hardware cores (e.g. decryption modules) are provided when using just the physical obfuscation layer because these are physical changes made to the FPGA, and the customized bitstream will work only with that particular FPGA. Additionally as will be discussed in relation to some embodiments below, at least one hardware core in the FPGA may be provided in combination with a logical key-based time-variant obfuscation layer.

[0051] In some embodiments, logical key-based and time-variant modifications are also made to the architecture. The modifications may be realized through the addition of permutation networks which modify the functions mapped to the FPGA. The time-variant logical-key may represent a portion of the unique identifier to the FPGA device in order to enable bitstream node-locking. In some embodiments, the time-variant logical-key may be pseudo-randomly generated. The time-variant logical-key effectively evolves the architecture of the programmable device with time during, for example, each time a device such as an FPGA is reprogrammed. Similar to physical-obfuscation, the vendor tool may make modifications to the bitstream at the end of the tool flow to implement the time-variant layer of obfuscation. For example, the tool will perform a series of obfuscation functions or transformations (e.g. permutations) on the configuration bits based on the unique logical key.

[0052] FIG. **4** is an illustrative diagram showing the mapping flow according to some embodiments. As shown in FIG. **4**, a device key K_D **401** is generated based on two portions **402** and **403** of the identifier **410** representing the physical and logical obfuscation layer, respectively. Each portion of the identifier **410** controls some aspect of the bitstream-to-device mapping via the device key **401** to generate a secure bitstream **404**. The secure bitstream **404** is mapped into the FPGA fabric **405**, including programmable interconnects **406** and lookup tables (LUTs) **407**. LUTs contain physical (fuse **408**-based) and time-variant (logical) selective inversion logic.

[0053] According to a non-limiting example, a multilayer transformation may be provided which operates on different portions of the bitstream in a serial fashion, such as 1) the LUT input ordering, 2) the LUT content ordering, and 3) block based transformation of the entire bitstream. FIG. **5***b* shows an illustrative example of a three level transformation scheme. A fourth level, which performs selective (key-based) inversion of the LUT contents, may be added after Level 2. In some embodiments, inclusion of the key-based inversion stage helps reduce the risk that functions like and with a truth table of 0001 may be used to deduce the transform key by observing the position of the "1". In some embodiments, these modifications to the bitstream are made in addition to, and with full knowledge of, the particular physical architectural changes already made to the device.

[0054] In some embodiments, the obfuscated and node-locked bitstream based on the unique device identifier is transmitted through an external interface to the authenticated FPGA.

[0055] In some embodiments, unlike the physical layer, additional hardware blocks are provided for the logical layer to perform the inverse transform. In one non-limiting example, for a multilayer transform structure, a set of three hardware cores perform serially the transform operations in reverse order of those performed by the Vendor tool. In this example, Levels 1 and 2 are both localized; that is, there are individual hardware modules which perform the inverse transform. Further according to the example, Level 3 is distributed along every row of the FPGA fabric; however, only some of these modules actually operate on data; the others may be "dummy" units which serve to further obfuscate the nature of the transform network. In this example, a successful Level 1 inverse transform may result in a valid bitstream; however, it may not function as expected unless the proper Level 2 and 3 inverse transform keys are applied.

[0056] FIG. **6***a* shows an illustrative example of a three level transformation scheme in the embodiments discussed above. In FIG. **6***a*, the Vendor tool transforms the bitstream using the three device-specific keys. Level 1 reorders the LUT inputs; Level 2 permutes the LUT content; and Level 3 performs a bit-level key-based bitstream permutation. In

the example in FIG. **6***b*, inverse-transforming occurs in reverse order using the appropriate inverse transform keys to recover the original bitstream. FIG. **6***c* shows an example Level 1 inverse transform network, operating on 16 bits of input, using 4 bits of key to transform data. Although three transformation levels and three inverse transform keys are shown in the example in FIG. **6***a*, any number of transform levels and any number of transform/inverse transform keys may be used to apply transformation to any of the FPGA resources. In some examples, a transformation level may apply selective inversion of a portion of LUT content bits based on the key, or selective inversion of a portion of LUT outputs based on the key, where the key can be physical or logical, or a combination of each.

[0057] Thus, with the combination of physical and logical architectural changes, the embodiments discussed above allow a unique bitstream-to-device mapping to be obtained. Though both physical and logical layers depend on a key, the physical changes may be accomplished using fuses, which cannot be changed at a later time. However, the logical key-based modifications may be time variant, which means that the architecture may effectively change with every reprogram cycle, making it impractical for an adversary to mount a known design attack.

[0058] FIG. **5***a* provides an illustrative diagram showing an embodiment of a device key management protocol. Responses from the PUF that are not retransmitted for authentication purposes may be used instead to generate the key, as shown in FIG. **5**. Furthermore, the responses used to generate the keys are selected by a decoder in the generation module; as an added measure of security, select bits may be randomly disconnected from the supply circuit using a series of fuses during enrollment.

[0059] A complete bitstream generation flow according to some embodiments is shown in the illustrative diagram in FIG. **3**(*b*). Each time the FPGA is upgraded, a different set of challenges may be issued, from which a different set of transform keys are generated. Such a moving target defense may help further secure the IP and prevent unauthorized reprogramming with previously used transform keys. Therefore, only after the device is authenticated and identified can the transformed bitstream be generated and sent to the device.

[0060] Having thus described several aspects of some embodiments of this invention, the following provides exemplary security analysis and overhead analysis of the device and method in the aforementioned embodiments comparing power, performance, and area overhead to commodity AES encryption cores.

[0061] Security Analysis

[0062] In some embodiments, a security analysis is provided for three attack scenarios, namely 1) brute force, 2) side channel attacks, and 3) destructive reverse engineering. The attacker may intend to reverse engineer the design either for monetary gain, or perform malicious modification and reprogram the device.

[0063] Brute Force Attack

[0064] A brute force attack represents the most challenging and time consuming attack on the system. Four attack stages are analyzed; for each stage, the attacker begins with incrementally more information.

Example Case 1.1.1

[0065] The attacker has, by some means, obtained a copy of the transformed bitstream.

[0066] Result: Without knowledge of the bitstream structure (e.g. fixed header contents), the attacker cannot identify the correct inverse transform key, even for Level 1. Thus, a brute force attack cannot be properly mounted, and the IP remains secure.

Example Case 1.1.2

[0067] The attacker has a copy of the transformed bitstream and knows the bitstream structure (e.g. typical contents of the header).

[0068] Result: The attacker can mount a brute force attack and attempt to deduce the Level 1 transform key. In this example, a 128 bit key may operate on 16 bit blocks, each of which is permuted using 4 bits. Thus, the number of possible permutations for each of the (128/4=32) blocks is $16^{32}=2^{128}$. This provides the first level of defense. Even if this is broken, Levels 2 and 3 are intact and the IF remains secure.

Example Case 1.1.3

[0069] The attacker begins with a Level 1 inverse transformed bitstream, and intends to break Levels 2 and 3.

[0070] Result: A Level 1 inverse transformed bitstream may be mapped to an FPGA or simulated using a bitstream-to-netlist tool. For each possible combination of the LUT inputs and outputs, the attacker performs the conversion, provides the proper stimuli, and observes I/O patterns. Without detailed knowledge of the intended functionality, or a sufficiently large set of test vectors, the process cannot be automated. Even with sufficient test vectors, brute force is not feasible: in an example of a set of 4×1 LUTs with four content bits and the possibility that some of the content bits may be inverted, the LUT can take 1 of L!×I possible states, where L is the LUT size, and I is the number of possible inversions.

[0071] I is computed as $\Sigma_{r=1}{}^{L}{}_{L}C_{r}$, which for L=4 gives 15 inversions; thus, each LUT can take 1 of 4!×15=360 combinations. Transforming the 4 bit LUT requires 2 bits of the key; thus, the 128 bit key operates on 64 blocks a search space of $360^{64}=2^{543.5}$. When considering the Level 3 transform, 2 transform bits may be provided, requiring 1 key bit, giving us up to 128 Level 3 inverse transformers. Depending on the size of the FPGA, only a portion of these may be used. With all 128 inverse transformers, this yields 21^{28} possibilities.

Example Case 1.1.4

[0072] The attacker has obtained all three transform keys, and has applied the Level 1 and 2 inverse transformers, leaving only the Level 3 transform intact.

[0073] Result: Without the architectural knowledge of which rows in the FPGA fabric have an active transformer, the attacker cannot know to which bits the Level 3 inverse transformer should be applied. Let R represent the number of rows in the FPGA fabric, and D the number of active inverse transformers. The possible permutations is represented by ${}^{R}P_{D}$. For a small FPGA (e.g. Xilinx XC3S50) with R=16 and D=12, we have ${}^{16}P_{12}\approx 2^{39.7}$ possible inverse transform networks. On a larger FPGA, with R=512 and

D=128, this would increase to $^{512}P_{128} \approx 2^{1127}$ possible networks. If D is unknown, these values represent the lower bound of attempts in a brute force attack.

[0074] Thus, in the example brute force attack scenarios discussed above, by itself, the Level 1 inverse transform presents a challenge to a brute force attacker; in the example case where the Level 1 inverse transform is compromised, Level 2, including the key-based inversion, and Level 3, including both the key-based input transform and the "dummy" inverse transformers make a brute force attack impractical.

[0075] Side Channel Attack (SCA)

[0076] Compared with brute force, a SCA is a more refined attack. Two example scenarios are presented herein in which one or more of the keys have been discovered in this manner.

Example Case 1.2.1

[0077] The attacker uses power analysis (e.g. DPA) to discover the challenge vectors stored in NVM.

[0078] Result: Responses are generated on-the-fly using a PUF, so leaking the challenge bits is not useful without an accurate PUF model. The generation procedure is purely combinational, using no latches of flip flops, and therefore is less vulnerable to power analysis.

Example Case 1.2.2

[0079] The attacker has discovered one or more of the CR pairs, for example through the use of wireless packet analysis.

[0080] Result: With sufficient CR pairs, the attacker may be able to refine a model of some kinds of PUFs (e.g. arbiter or ring oscillator PUF), making the choice of PUF crucial to system security. In some embodiments MECCA PUF may be a good choice because it is resistant to these attacks. In any case, very few pairs are sent each upgrade, limiting the attacker's potential knowledge of the system.

[0081] SCA attacks may be used to leak the Challenge vectors or isolate CR pairs from packet analysis. However, as discussed above in Example case 1.4 under the Brute Force Attack scenario, knowledge of the Level 3 key is insufficient to fully inverse transform the design. Thus, in the example SCA scenarios discussed above even if modeling attacks are successful, the IP remains secure.

[0082] Destructive Reverse Engineering (DRE)

[0083] DRE is an expensive and time consuming process, but it can reveal the inner workings of the device. Two example scenarios of using DRE attacks are discussed.

Example Case 1.3.1

[0084] DRE is used to reveal the structure of the Level 3 transform network, including which rows contain deactivated inverse transformers.

[0085] Result: This reduces the number of possible bitstream permutations. However, without further analysis (e.g. successful PUF modeling), the IP remains secure.

Example Case 1.3.2

[0086] DRE is used to reveal the PUF structure, potentially making the device vulnerable to these attacks and reducing the search space for the correct transform key.

[0087] Result: Modeling attacks have been proposed and successfully executed for certain PUFs (e.g. Arbiter PUF [Ref. 12]). Nevertheless, there is inherent uncertainty in the probabilistic approach employed by the attack models, and some PUFs have been proposed [Ref. 13, 14] which are resistant to these attacks. Even if the transform key is revealed, knowledge of the Level 3 transform network, which may demand further DRE, is desired to make use of it.

[0088] Therefore, from the above analysis of three types of example attack scenarios, it is clear that even with a combination of SCA and DRE attacks, some level of brute force is still necessary to inverse transform a single bitstream for a single device. Of all the attacks presented above, the only one with wide-ranging consequences is the discovery of the Level 3 transform network. By itself, this does not fully compromise the system; significant analysis, and some brute force, may still be required. Furthermore, the device-specific keys and CRCP disclosed in some embodiments also ensure that unauthorized reprogramming on other IoT connected devices will not be possible, since only one specific device can acquire the targeted upgrade, making malicious modification and reprogramming infeasible. This approach reduces, and perhaps entirely mitigates, the economic motivation for an attacker.

[0089] 2) Overhead Analysis

[0090] In this section, the power, performance, and area overhead incurred using the bitstream security system disclosed in some embodiments are analyzed. Components are implemented in Verilog, simulated to verify functionality, and synthesized with Synopsys Design Compiler using a 90 nm cell library. Results for Area, Power, Delay, and Energy of the various modules are listed in Table 1. Results represent an FPGA with one Device Key Module (DKM), three Response Generator Modules (RGM), one Level 1 and one Level 2 Inverse transform Logic Module (DLM1 and DLM2), and 32 DLM3 modules.

TABLE 1

Synthesis results at 90 nm. "Num Inst." is the number of instances considered in the results. Delay and Energy are for a 512 kB bitstream.

Mod. Name	Num Inst.	Area (μm2)	Area (Gates)	Pow. (mW)	Delay (ns)	En. (pJ)
DKM	1	9398	827	1.08	1.38	1.49
RGM	1	145	34	0.02	1.18	0.02
DLM1	1	1063	115	0.18	6200	1120
DLM2	1	4273	406	0.77	33.0	25.4
DLM3	32	4328	460	0.67	0.17	3.64
Total		19207	1842	2.72	6236	1150

[0091] 2.1) Device Key Modules

[0092] In this example, the DKM is a purely combinational circuit with no memory elements. The input selects 2 of 8 PUF-generated responses, each 64 bits in length.

[0093] 2.2) Response Generator Modules (RGMs)

[0094] In this example, the RGMs are based on the MECCA PUF [Ref. 13], which uses an existing SRAM memory array to generate a response. A programmable pulse generator using a tapped inverter chain interfaces with existing SRAM peripheral logic; very little extra hardware may be needed.

[0095] 2.3) Inverse Transform Logic Modules

[0096] In some embodiments, inverse-transformation may occur in three separate stages, each controlled by a separate

128 hit key. Note that timing is reported for each module independent of external factors, such as serial to parallel (or parallel to serial) conversion in and out of the modules.

[0097] 2.3.1) Example with Level 1: In this example, a 16 input Banyan switch network implements the Level 1 inverse-transformation logic. Four bits of the transform key are used as inputs to each column of switches.

[0098] 2.3.2) Example with Level 2: The second level inverse transforms the LUT content Like Level 1, the key determines the mapping from input to output ordering. In this example, LUT responses are defined by 4 bits; thus, the network operates on 16 inputs, each a 4 bit vector. Selective inversion of the transform bits is determined by the transform key.

[0099] 2.3.3) Example with Level 3: The third level inverse transforms the LUT inputs, and inverse transformers are distributed among the rows in the FPGA fabric. An immense FPGA fabric is provided in this example with 1024 rows, and therefore 1024 transform networks (some are deactivated). All LUTs are 4×1 in this example, and thus have two select inputs.

[0100] 3) Comparative Analysis

[0101] The total area, power, and latency overhead may be analyzed in the embodiments disclosed above as the sum of the respective parameters for each module. Table 2 compares the analysis results with several AES cores (from both IP vendors and literature).

TABLE 2

Comparing the Node Locked Bitstream (NLB) with AES ASIC cores. Delay and Energy are calculated from throughput for a 512 kB bitstream.

Mod. Name	Tech (nm)	Area (Gates)	Pow. (mW)	Delay (μS)	EDP (J*s)
NLB	90	1.8k	2.72	6.2	1.07e−13
[Ref. 15]	180	<3k	—	64000	—
[Ref. 16]	130	3.1k	5.62	33850	6.44e−6
Tiny [Ref. 17]	130	<5k	—	40960	—
Std. [Ref 18]	90	8.8k	—	2800	—
Std. [Ref 17]	130	<9.5k	—	630	—

[0102] Table 2 shows that in some embodiments, even after scaling power and throughput to the 90 nm node, the Node Locked Bitstream method is faster than the area- and power-optimized crypto cores, and incurs a lower area and power overhead, making it ideal for power- and area-constrained systems. Furthermore, like the crypto cores, it offers excellent security against brute force attacks. In addition, it is more resilient to SCA and even DRE attacks.

[0103] The NLB system disclosed herein is capable of protecting FPGA bitstreams against a number of attacks, including brute force, side channel, known design attacks and destructive reverse engineering, effectively preventing IP piracy and malicious modification. Having thus described several aspects of some embodiments of this invention, it is to be appreciated that various alterations, modifications, and improvements will readily occur to those skilled in the art.

[0104] For example, the NLB concept may be extended, first by adding additional layers of security beyond those previously listed for FPGA, and by applying these concepts to the domain of software security for microcontrollers (firmware) and more complex processors (full software applications, including those compiled to machine language or interpreted code, for example Java). These extensions are attractive for a number of reasons:

[0105] Additional security makes it less likely for an attacker to successfully pirate, reverse engineer, or maliciously modify the IP by including terms which exhibit factorial growth.

[0106] It allows for the consideration of additional FPGA hardware structures, and presents opportunities to identify more cost effective modifications, providing equivalent-or-better security using the same or fewer key bits; this in turn provides an empirical means to optimize security versus area/power/delay overhead in different FPGA implementations.

[0107] The inventors have recognized that microcontrollers (and their various application domains, including automotive, communication, consumer electronics, among others) present an even larger market than FPGA, and receive firmware upgrades at least as frequently as an FPGA-based device from trusted vendors (e.g. Original Equipment Manufacturers, OEM). Ensuring the integrity of these firmware upgrades, especially those transmitted Over the Air (OTA) is essential to maintaining device security.

[0108] A discussion of microcontroller firmware security further leads to methods which can improve security for systems with more complex General Purposes Processors (GPPs), including desktop and laptop computers. Users of these systems can download software from a plethora of online sources, many of which can be counterfeit or malicious, resulting in malware which can wreak havoc on a system or leak personal information to an attacker. Controlling the sources of these applications and judiciously restricting the ability of a target architecture to execute them can help curb both the distribution of malicious software, as well as the unauthorized distribution of proprietary software, thus doubling as an alternative to software node-locking.

[0109] The following three sections describe additional embodiments providing extensions to the NLB framework discussed above for the application in (1) FPGA bitstream security. (2) microcontroller firmware security, and (3) general purpose processor security.

[0110] Extensions of NLB for FPGA

[0111] In some embodiments, FPGA security can be extended using additional permutation and selective inversion networks, operating not only on the LUT content, LUT input, and the bitstream as a whole, but on any amenable hardware structure on the FPGA. These resources include, but are not limited to, the following: configurable logic blocks (CLBs), routing/programmable interconnects, block RAM/embedded memories, DSP blocks, IO blocks and clocks/PLLs.

[0112] A simplified example of the FPGA architecture combining the mentioned resources is shown in FIG. **7**. Tables 3, 4 and 5 summarize different aspects of implementing the obfuscation model on different resources according to some embodiments. The NLB model may be implemented on individual resources, or on multiple resources in parallel to increase the level of security.

TABLE 3

Various aspects of implementing permute and selective inversion networks on CLB resources.

Resource	Sub-resource	Resource Description	Architectural change required to map the IP from obfuscated bits	Required Key bits	Resultant Diversity
CLB content (FIG. 8)	LUT Content	Lookup Tables (LUTs) contain SRAM cells which ? old function responses ("Current") required for the design.	The actual content bits to the configuration bitstream will be permuted using the compilation tool. In the FPGA, a harware block within the LUT undoes this operation. Forward and inverse transforms are done using a key.	Assume number of LUT inputs is 1 and number of Content bit ? = 2?. The required key to shuffle L bits is $\log_2(L)$. Example: for 4 input LUT with 16 content the key size is $\log_2(16) = 4$.	For LUT with ?, ?the number of different possibilities would be I?. Example: Let ? = 4, there are 4? (24) possible combinations. In practice, ? = 2? or 2? are more common.
	LUT Content Selective Inversion		Certain content bits will be inverted inside the tool based on a Key. Symmetric inverse transform ? recovery of original key bits. The inversion logic take the key and inverts based on ?. The resultant bits in the SRAM cell maps the original design.	To invert, one key bit is required per content bit. Key size equals LUT size.	For a certain LUT. the number of content bits to be ? is equivalent to the number of logic ?'s in the subkey, given ? r. Attackers must search all possible values of ?, requiring Σ? ^{L}C? Example: Let L = 4. This gives 1? possible combinations. LUTs where L = 2? or 2? are common in ?, ? large search spaces.
	Funtion Input Multiplier	LUT function evaluation results from the selection of certain content bits being selected by a multiplexor (mux). the mux inputs represent function inputs. These can be selectively modified.	One hardware block performs the inverse transform on the function input, resulting in correct function output from the LUT.	Requires ? = $\log_2(L)$ key bits to permute the inputs for ? LUT with ? funtion responses.	For a LUT of ? inputs, there can be ? possible orderings. Example: for ? 4 input LUT, an attacker ? to consider 4? = 24 different possibilities.
CLB content	FF-Mux bit inversion	Content bits in LUTs only implement combinational logic. To map sequential logic, Flip Flops (FF) are needed. A mux selects if the LUT output will be connected with the FF.	A single bit in the configuration bitstream is responsible for the FF selection via MUX. The select bit of the MUX that ? bypasses the FF can be ?	The selection of FF is done by a 2:1 MUX which has one select bit. The key size is therefore 1 for each ?.	For each LUT, 2 different probability. Either the LUT goes to the FF, or bypasses the FF.
CLB content	LUT output inversion	The final LUT output ? with or without FF) can be inverted. This outpu? connect to the inputs of multiple LUTs.	For a single LUT, one inversion logic is required with the output. Based on the key, the output will be inverted.	1 Key bit required for a single output.	For any LUT, 2 different probabilities are present. However, this effects other LUTs that take this output as an input. Therefore the search space increases. If the output Y is input to some other LUT; while ? each possible ? of the connected LUT, the adversary has to consider both Y and Y?,
CLB content	Carry logic Mux bits inversion	Carry logic is available inside CLBs with each LUT fo? propagation of carry bits while ? long digits.	Carry logic of LUT is selected ? MUX. For 2:1 MUX the selection bit is a single ?. This single configuration bit can be altered/ inverted using one inversion logic.	Only 1 Key bit is required per LUT.	for each LUT the design can either have or not have ? logic based on the key bit. For N number of LUTs the chances are 2^N
CLB content	Inter-connect matrix inside CLB	? channels (wires) go inside the CLB and connect to LUTs. LUT outputs also connect to the input of adjacent LUTs of the same CLB or feedback to itself. Such connections are done by an interconnect matrix inside the CLB.	To our knowledge the low level architecture of the interconnect matrix is not revealed by the ?. However, it should be similar to Switch ? architecture which is known. Therefore we can refer to the analysis of the Switch Box.	Refer to the analysis of the Switch Box.	Refer to the analysis of the Switch Box.

? indicates text missing or illegible when filed

TABLE 4

Various aspects of implementing permute and selective inversion networks on routing resources.

Resource	Sub-resource	Resource Description	Architectural change required to map the IP from obfuscated bits	Required Key bits	Resultant Diversity
Routing resources outside CLB (FIG. 9)	Connection box	Connection boxes connect wires to and from CLBs with the main channel outside the CLB.	Refer to the analysis of the Switch Box.	Refer to the analysis of the Switch Box.	Refer to the analysis of the Switch Box.
	Switch Box	The Switch boxes connect horizontal and vertical routing channels. Each Switch Box is composed of a number of switch points which can connect certain wires. The low level design is shown in the ⍰. Based on the configuration bits the switch point routes certain wires to different directions. Inside the switch points, SRAM cells connect with the MUXs and tristate buffers that control the routing. These cells hold the configuration bits for the switch points.	There are 12 configuration bits for each switch point. If the bits are shuffled, 12 bits would require a deshuffler block controlled by 4 key bits. If the bits are inverted inside the tool, the inverted configuration bits have to pass through the inversion logic before programming the switch point. As there are multiple switch point per switch box, and a large number of switch boxes inside the FPGA, we may obscure only a selected number of switch boxes. It will keep the key size limited and improve the difficulty of deobfuscation.	For a single switch point with B configuration bits, N switch points in a switch box, and the S different switches to consider, then total key bits required for shuffling would be, $N * S * Log_2(B)$. For inversion. If r bits are inverted the required key for whole FPGA would add a factor of r bits to the key.	For shuffling, the possible search space is B⍰ for switch point. If r bits are inverted among B, the search space is $= \Sigma_r {}^{B}{}^{B}C_r$ If both ⍰ and inversion are done, the search space increases to B⍰ $\Sigma_r {}^{B}{}^{B}C_r$ for a single point. Therefore, for the whole FPGA it is $N + S + B$⍰ $\Sigma_r {}^{B}{}^{B}C_r$.

⍰ indicates text missing or illegible when filed

TABLE 5

Various aspects of implementing permute and selective inversion networks on BRAM & DSP

Resource	Sub-resource	Resource Description	Architectural change required to map the IP from obfuscated bits	Required Key bits	Resultant Diversity
Block RAM	RAM Content	Embedded block RAM are actually kilobytes of SRAM for storing data. These RAMs are hard blocks and can be initialized in different sizes and operational modes which is defined in the bit stream. The block RAM content, the programmable interconnects, and the specifications are defined by specific groups of bits in the bitstream frame. A sample frame is shown in FIG. 10.	If the initial contents of the RAM are shuffled or inverted inside the tool, the inverse transform can be applied internally using shuffle blocks and inversion logic. However, if the content bits of the SRAM are readable while the FPGA is operating, the adversary may be able to exploit this to determine the shuffling pattern. Therefore, it may be more secure to not modify the memory configuration if there is also an external memory interface.	Valid assumption depends on details of the bitstream used for configuring Block RAMs.	
	RAM Size (8 KB, 36 KB etc.) Data width and address width ⍰ made (⍰/Single) Multi-RAM Interconnect Logic Read/Write Operation Sequence Specification Interconnects		Operational mode and RAM size are defined while writing the HDL code of the IP which turns into configuration bits. These bits are placed into specific frames. The exact frame structure which shows exactly which bits are responsible for certain specification is not open to the public. But as the vendors have the information, they can shuffle those bits and later deshuffle them using a centralized deshuffler inside the FPGA		
DSP Blocks	Bits specifying the function to be performed Interconnects	Dedicated hard DSPs in the FPGA are available. For example, ⍰ Cyclone⍰ and Xilinx Virtex-⍰ Pro devices contain embedded 18 × 18-bit multipliers, which can be split into 9 × 9 bit multipliers. Xilinx	In some of the Xilinx DSP block, various combination of control inputs prepare the DSP slice to perform certain operations such as addition, subtraction, and multiplication. Similar to block RAM the various operational mode and interconnects of the block that is	Valid assumption depends on details of the bitstream used for configuring DSP blocks.	

TABLE 5-continued

Various aspects of implementing permute and selective inversion networks on BRAM & DSP

Resource	Sub-resource	Resource Description	Architectural change required to map the IP from obfuscated bits	Required Key bits	Resultant Diversity
		Vitrex-5' XirameDSP slices contain a dedicated 18 × 18-bit 2?s complement signed multiplier,? logic, 48-bit accumulator, and pipeline registers.	written in the HDL is defined in the bitstream and the exact locations of the bits are vendor specific secrets. But vendors can utilize our obfuscation model as the bitstream format details for any resource are available to them.		
Clocks and I/O		Not implemented. Clocking can be easily measured through side channels, I/O direction can be directly measured, and improper I/O can result in physical damage to the board.			

? Resultant diversity refers to the number of possible? introduced by the?.? is a practical imp?, the? of? will be significantly greater than the examples given here (due to exponential and fac? growth). Furthermore, these techniques are applied design-wide, and will therefore effect hundreds or thousands of different? depending on the size of the design.
? indicates text missing or illegible when filed

[0113] Resource Ranking:

[0114] Based on analysis from Tables 3, 4 and 5 the combination of LUT content transformation and LUT content random inversion is a preferred means of obfuscation that is very effective. This can also be an effective way to prevent bitstream tampering in some embodiments as an attacker would be unable to figure out the functionality of the bitstream by observing how the bits get stored into the SRAM cells. Only the proper key can reveal how the bits finally execute in a running FPGA. In some embodiments, transformation or inversion of switch box resources can also obfuscate the original IP to a great extent because routing resources cover a major portion of the programmable fabric. However, only altering routing bits might not be sufficient as the LUT bits can contain significant information about the IP. Therefore, an adversary might be able to partially reverse the IP even though the routing is obfuscated. A powerful solution would be randomized transformation and inversion of both routing resources and LUT contents. Obfuscation of embedded BRAM and DSP can be explored further if more information about the bitstream variations for different resource settings are available (e.g. by the FPGA vendor).

[0115] Demonstration on Test Framework:

[0116] In one embodiment, a software demonstration of the NLB techniques is provided using VPR, an academic tool which performs Verilog-to-FPGA mapping for test FPGA frameworks. The tool can take as input either a Verilog HDL circuit, or a circuit described in the Berkeley Logic Interchange Format (BLIF), as well as runtime parameters defining the key length and how the key is partitioned among the different hardware structures. In a non-limiting example, the tool outputs the following:

[0117] A "gold standard" structural Verilog file for functional simulation of the mapped design. This design uses the original primitives (e.g. 4, 5, or 6 input LUTs) to realize the circuit functionality.

[0118] A Verilog file that uses the modified primitives implementing key-based permutation and selective inversion used to realize the secure FPGA. Subkeys are passed as parameters to individual LUTs. This file can be used to functionally verify the design against the gold standard.

[0119] Two bitstream files, comprised of the LUT contents of the design. These are used to compare the similarity between the two bitstreams using the Hamming Distance metric.

[0120] A Key file stores all subkeys used in the secure design. The size of this key is used to compute the overhead in bitstream size.

[0121] A security metric based on the theoretical formulation

$$S = \sum_{r=1}^{L} \binom{L}{r} \times L!,$$

representing an empirical measure of security for LUT-only obfuscation. This enables design space exploration of tradeoffs between key length, key partition methodology, and relative security, as well as optimization of these parameters for different designs and FPGA platforms.

[0122] The output Verilog files can be simulated using ModelSim, VCS, or similar Verilog simulation application. In one embodiment, a testbench can be written to compare outputs between two modules (e.g. gold+secure (with correct key) or gold+secure (with incorrect key), demonstrating the architectural specificity of the respective bitstreams.

[0123] (2) Extensions of NLB for Microcontroller Security

[0124] A bitstream may generally refer to a stream of binary bits, such as those in a binary file used for programming the firmware of a microcontroller. For microcontrollers, the firmware-securing protocol is nearly identical to that of the FPGA bitstream security. This is because the firmware source (e.g. the device vendor) is inherently trusted, and the firmware will generally be compiled (rather than interpreted via virtual machine, for example). Just as in the FPGA Node Locking framework, the combination of key-based permutation and selective inversion may be used to provide effective architectural diversification in some embodiments. According to an aspect, the framework similarly relies on a set of challenge vectors sent by the OEM to the device, and uses the responses (generated by PUF) to identify the device. The binary is permuted individual bits are selectively inverted using multiple key-based hardware networks, affecting the instruction decoding, the program counter/control flow, functional units (e.g. barrel shifter/multiplier/floating point, etc.), and potentially any other available structures. At the hardware level, the reverse operations may be performed using the internally-generated key(s) just-in-time for execution. Therefore, in some

embodiments this method incurs a small, one time overhead when the firmware loads, and a small overhead during execution in the decode stage.

[0125] (3) Extensions of NLB for CPU Security

[0126] For general software application security, a different protocol may be used because the myriad software sources are not necessarily trusted, and many programming languages do not rely on compilation to machine code (e.g. Java bytecode). Therefore, in some embodiments a system may be provided whereby applications are hosted in a trusted source, which modifies the executable/bytecode/intermediate language/etc. in such a way that only one system will be capable of properly executing the code. An exemplary system flow for general application software is pictured in FIG. **11**. In one embodiment, the user is only able to download programs from a set of one or more trusted servers. Applications which are hosted in this trusted space may be vetted, scanned, and verified to be safe.

[0127] In some embodiments, users wishing to download a program may simply request to download the application from the server as usual. Over a secure channel the server transmits challenge keys, which are generated locally using a hardware PUF and secured prior to transmission. Once identified, a random key is selected from the user's set of keys (stored on the cloud) and uses it to modify the application binary, which renders it unexecutable for any system except the system making the download request. The application may then be downloaded from the server and installed on the user's machine as usual. In some embodiments, the application files are stored in their modified format, so that the application cannot be transferred to another system, thus effectively node-locking the program without relying on other authentication methods (e.g. USB drive with key file, MAC address authentication, licensing server, etc.). According to an aspect, the cost introduced for the software supplier and the user is relative low compared to the level of security offered and potential for more secure node-locking of proprietary software made possible by this method. Additionally, use of the trusted cloud server and trusted developer tools may provide interoperability and backwards compatibility with existing code bases.

[0128] In some embodiments, independent software development (e.g. for hobbyist developers, students, etc.) may be facilitated by this framework. When developing an application, a user may compile the binary for their particular system using typical methods (e.g. GCC); the application binary will be transformed using a temporary key, which is generated for each application and allows that application to run on that system alone. Cloud development tools and platforms (e.g. Microsoft Azure) can potentially integrate these capabilities according to some embodiments.

Additional Example

[0129] In this example, a low-overhead FPGA bitstream obfuscation solution is presented that can maintain mathematically provable robustness against major attacks. The solution exploits the identification of FPGA dark silicon, i.e., unused LUT memory already available in design mapped to FPGAs, to achieve bitstream security. It helps to drastically reduce the overhead of the obfuscation mechanism. The approach does not introduce additional complexity in design verification and incurs a low performance and negligible power penalty. In particular, the mechanism described here permits the creation of logically varying architectures for an FPGA, so that there is a unique correspondence between a bitstream and the target FPGA. FIG. **12** shows a high-level overview of this approach. Compared to existing logic obfuscation techniques, no design-time changes to the FPGA architecture or expensive on-chip public key cryptography is required. In addition to obfuscation of design functionality, our approach also enables locking a particular bitstream to a specific FPGA device, helping to prevent piracy of the valuable IP blocks incorporated in a design. Therefore, it goes well beyond standard bitstream encryption in FPGA security. Furthermore, it is targeted to the protection of FPGA bitstreams, rather than hardware metering of integrated circuits. Finally, the procedure seamlessly integrates into existing CAD tool flows for programming FPGA devices

[0130] The typical island-style FPGA architecture consists of an array of multi-input, single-output lookup tables (LUTs). Generally, LUTs of size ii can be configured to implement any function of n variables, and require 2^n bits of storage for function responses. Programmable Interconnects (PIs) can be configured to connect LUTs to realize a given hardware design. Additional resources, including embedded memories, multipliers/DSP blocks, or hardened IP blocks can be reached through the PI network and used in the design.

[0131] The nature of FPGA architecture requires that sufficient resources be available for the worst case. For example, some newer FPGAs may support 6 input functions, requiring 64 bits of storage for the LUT content. However, typical designs are more likely to use 5 or fewer inputs, while less frequently utilizing all 6. Note that each unused input results in a 50% decrease in the utilization of the available content bits. This leads to an effect that resembles dark silicon in multicore processors, where only a limited amount of silicon real estate and parallel processing can be used at a given time. To make this analogy explicit, we refer to the unused space in FPGA as "FPGA dark silicon". Note that in spite of the nomenclature the causes behind dark silicon in the two cases are different. For multicore processors, it is typically due to physical limitations or limited parallelism; for FPGAs, it is the reality of having sufficient resources available for the worst-case which may occur infrequently, if at all.

[0132] Our approach depends on the presence of FPGA dark silicon to be exploited for obfuscation needs. Consequently, we made a comprehensive evaluation of this phenomenon to identify the scope and scale of this phenomenon. Table 6 shows the result of this evaluation. Note that the evaluation uses benchmark designs of diverse scale and complexity, taken from three publicly available benchmarks, e.g., EPFL Arithmetic Benchmark Suite (http://lsi.epfl.ch/benchmarks), Opencores (http://opencores.org), and Github (http://github.org). All benchmarks were mapped to an Altera Cyclone V device [1]. The Cyclone V contains two 6-input Adaptive LUTs (ALUTs) per Adaptive Logic Module (ALM), and 10 such ALMs per Logic Array Block (LAB).

[0133] Our evaluation shows the availability of significant unused space across the diversity of benchmarks. Even for small combinational circuits (less than 2000 LUTs), roughly 50% of the LUTs mapped use 4 inputs or fewer, while 82% of the LUTs mapped use 5 inputs or fewer. The effect is more pronounced for large sequential benchmarks, where 69% of LUTs are 4 inputs or fewer, and 82% use 5 inputs or fewer.

TABLE 6

CUMULATIVE PERCENTAGE OF 1-7 INPUT LUTs

Circuit	Cumulative % of LUTs with Inputs n						Total
Name	≤2	3	4	5	6	7	LUTs
alu4	10.6	26.1	48.4	77.7	97.9	100	188
apex2	11.4	26.0	52.3	91.0	99.1	100	669
apex4	16.7	27.4	50.3	89.4	97.6	100	574
ex5p	41.0	42.1	58.7	84.5	98.4	100	373
ex 1010	16.9	24.2	46.4	84.8	98.3	100	711
misex	14.0	27.7	46.9	84.0	97.5	100	480
pdc	16.3	28.5	51.9	77.7	98.4	100	1588
seq	16.6	51.9	51.9	89.1	99.0	100	727
spla	17.8	53.1	53.1	79.9	98.7	100	1509
Avg.	17.9	29.0	51.1	84.2	98.3	100	758
div	7.8	13.1	32.7	60.1	100	—	12.4 k
hyp	0.9	28.8	42.6	64.0	100	—	45.3 k
log2	7.0	17.2	39.5	59.7	99.0	100	7894
mult	2.5	25.0	50.5	59.0	99.0	100	5553
sqrt	5.8	5.0	43.5	84.5	100	—	3685
square	5.6	55.9	60.2	74.6	100	—	4066
Avg.	4.5	24.2	44.8	67.0	99.7	100	13.1 k
AES	39.7	64.2	71.0	100	—	—	4112
AOR32	20.7	22.9	31.5	46.8	97.8	100	2299
BTCM	32.5	95.3	99.8	100	100	—	41.0 k
JPEGE	45.2	37.6	48.4	67.0	99.4	100	5154
Salsa20	59.9	57.4	93.8	93.9	100	—	2836
Avg.	39.2	55.5	69.1	81.5	99.4	100	11.1 k

[0134] To quantify the role of dark silicon, we define a metric, the Occupancy of the FPGA, as the percentage of content bits used per LUT, divided by the total number of available bits in the LUTs which are used. We use the Cyclone V device architecture as a case study. In Eqn. 1, the number of n-input LUTs (# (LUTn)) is multiplied by the content bits used for that LUT (2^n); this value is divided by the LUT capacity **2'** times the number of LUTs used in total; the variable p indicates the maximum power of the LUT, which in this case is 6. This yields the ALUT Occupancy. Next, ALM Occupancy is computed in Eqn. 2 as the average number of ALUTs per ALM; in this case, the ALM_MAX_CAP is 2. Finally, the LAB Occupancy is computed in Eqn. 3 as the average number of ALMs per LAB; LAB_MAX_CAP is 10 for the Cyclone V. Finally, the product of these three terms gives the overall occupancy (Eqn. 4), indicating the true percentage of fine-grained resource utilization at the content bit level for the given FPGA architecture.

$$O_{ALUT} = \frac{\sum_{n=1}^{p} \#(LUTn) \times 2^n}{\#(LUT) \times 2^p} \quad \text{(Eqn. 1)}$$

$$O_{ALM} = \frac{\#(ALUT)}{\text{ALM\_MAX\_CAP} \times \#(ALM)} \quad \text{(Eqn. 2)}$$

$$O_{LAB} = \frac{\#(ALM)}{\text{LAB\_MAX\_CAP} \times \#(LAB)} \quad \text{(Eqn. 3)}$$

$$O_{Total} = O_{ALUT} \times O_{ALM} \times O_{LAB} \quad \text{(Eqn. 4)}$$

[0135] We computed O_{Total} for a set of 9 combinational benchmark circuits and found the average occupancy to be 26%±4%, leaving nearly ¾ of the available content bits within the used LUTs empty. This same phenomenon may extend to designs that require more resources, e.g. large arithmetic circuits for which the occupancy is slightly higher (31%±4) and the previously listed IP cores, for which the occupancy is significantly lower with higher variance (12%±8).

[0136] A. Bitstream Protection Methodology

[0137] In this section, we describe a bitstream protection methodology in accordance with an embodiment and its integration into the design flow.

[0138] A.1 Design Obfuscation

[0139] As described above, most of the LUTs used to implement a given design do not require full utilization of the available memory bits. This leaves open spaces where additional function responses can be inserted to obfuscate the true functionality of the design, which in turn makes it more difficult for an adversary to make a Targeted Malicious Modification.

[0140] For example, consider a 3-input LUT, which contains 8 content bits, used to implement a 2 input function, Z=X∀Y. A third input K can be added at either position 1, 2, or 3, leaving the original function in either the top or bottom half of the truth table, or interleaved with the obfuscation function. An example of this is shown in the 4 LUT design of FIG. **13**, as well as in Table 7. In this case, the correct output is selected when K=0; if K=1, a response from the incorrect function (Z=X∧Y) is selected. However, if it is not known that this truth table is obfuscated, the function could possibly be $Z=XYK \vee X\overline{Y}\overline{K} \vee \overline{X}\overline{Y}K$, $Z=XYK \vee \overline{X}\overline{Y}K \vee \overline{X}\overline{Y}\overline{K}$, or $Z=XYK \vee \overline{X}\overline{Y}K + \overline{X}Y\overline{K}$—three functions with distinctly different responses.

TABLE 7

EXAMPLE LUTs WITH 2 PRIMARY INPUTS AND 1 KEY INPUT, THE TRUE FUNCTION IS Z = X ⊕ Y, WHICH IS ONLY SELECTED WHEN K = 0.

X	Y	K	Z	X	K	Y	Z	K	X	Y	Z
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0	0	1	1	0
1	0	0	1	1	0	0	1	1	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0
1	1	0	0	1	1	0	0	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1
		(a)				(b)				(c)	

[0141] The security of this approach depends on the number of LUTs that are mapped for a given design; with more LUTs obfuscated in this manner, the security increases dramatically. For real-world designs, this is not likely to be a limitation, since designs will typically implement several hundred to several thousand device resources. Further analysis of this security is presented in Section B.3.

[0142] A.2 Key Generation

[0143] The first step for the secure bitstream mapping is a low-overhead key generator, such as a nonlinear feedback shift resister (NLFSR), which is resistant to cryptanalysis. A Physical Unclonable Function can also be used; though this requires an additional enrollment stage for each device, it has the added benefit of not requiring key storage. Various PUF-based key generators have been proposed, including PUFKY, which are amenable to FPGA implementation. Furthermore, using a PUF-based key generator requires that FPGA vendor tools provide floorplanning and/or enable assignment to specific device resources for reproducibility. In general, we refer to the key generator as the system's

CSPRNG, or cryptographically secure pseudorandom number generator. The specific CSPRNG used depends on the application requirements.

[0144] A.3 Initial Design Mapping

[0145] The second step is the synthesis of the HDL design into LUTs. In some embodiments, this can be performed by freely available tools such as ODIN II; it is also possible to configure commercial tools, e.g. Altera Quartus II, by including specific commands into the project settings file (*.qsf) before compilation; this generates a Berkeley Logic Interchange Format (BLIF) file with technology-mapped LUTs. It should be appreciated that the implantation of the second step is not limited to the above mentioned methods and any suitable tool and/or file format may be used.

[0146] A.4 Security-Aware Mapping

[0147] The security-aware mapping leverages FPGA dark silicon (Section A.1) for key-based design obfuscation. The software flow is shown in FIG. **14**. The following is a brief description of the processing stages:

[0148] 1. Analysis: Inputs to this stage include the BLIF design, as well as the maximum size of LUT supported by the target technology. The circuit is parsed, analyzed, and assembled into a hypergraph data structure. The analysis also determines the current occupancy.

[0149] 2. Partitioning: Inputs to this stage include the hypergraph data structure, as well as the key length. The hypergraph is partitioned into a set of subgraphs which share common inputs/outputs using a breadth-first traversal. Nodes are marked as belonging to a particular subgraph such that those with the greatest commonality are grouped into partitions. The number of partitions is directly proportional to the size of the key.

[0150] 3. Obfuscation: For a device supporting k-input LUTs, every LUT with at most (k−1)-inputs is obfuscated by implementing a second function using the unoccupied LUT content bits. One additional input is added to the LUT which corresponds to the key bit used to select the correct half of the LUT during operation. The second function can be either template-derived, such as basic logic operations (nand, nor, xor, etc.), or functions implemented in other LUTs in the same design.

[0151] 4. Optimization: In this stage, individual LUTs are optimized using the Espresso Logic Minimizer. The optimized Espresso output is converted back into the internal representation. This process significantly reduces both the output file size, as well as eventual compilation time in the FPGA mapping tool.

[0152] 5. Output Generation: The output file generation can take one of two formats: (a) structural Verilog, which implements the circuit as a series of assignment statements, or (b) using device-specific LUT primitive functions. The second option is preferred because using low-level primitives ensures that the design will be mapped with the specified LUTs.

[0153] The number of LUTs per partition is an especially important metric, as it has a direct impact on both the overhead and the level of security. Furthermore, the partitioning and sharing of key bits need to be done judiciously, as a random assignment can potentially dramatically increase area overhead (see Section B.2). Thus, key sharing, when paired with the LUT output generation, is intended to (a) reduce overhead, and (b) strongly suggest to the physical placement and routing algorithms used by the commercial mapping tool to group certain LUTs in a given ALM and/or LAB, and thus minimize area overhead. Ideally, this process could be integrated into a commercial tool itself to enable technology-dependent optimizations.

[0154] A.5 Communication Protocol and Usage Model

[0155] The security-aware mapping procedure creates a one-to-one association between the hardware design and a specific FPGA device, since selection of the correct LUT function responses depends on the CSPRNG output. This means that OEMs must have one unique bitstream for each key in their device database. Therefore, it is critical that the correct bitstream is used with the correct device. Modern FPGAs contain device IDs which can be used for this purpose; alternatively, if a PUF is used as the CSPRNG, the ID can be based on the PUF response. Using existing FPGA mapping software, generating a large number of bitstreams will take considerable time; however, with modifications to the CAD tools, the security-aware mapping can be done just prior to bitstream generation, so that the design does not need to be rerouted.

[0156] The initial device programming, prior to distribution in-field, may be done by a (potentially untrusted) third party. The third party is able to read the device ID, but does not require access to the key database. Similarly, device testers do not need access to the key, merely the ability to read the ID. This allows OEMs to keep the ID/key relation secret. Once the device is in field, the remote upgrade procedure differs slightly from the initial in-house programming. The typical upgrade flow is shown in FIG. **4**. After finalizing the updated hardware design, it is synthesized using the security-aware mapping procedure. Target devices are queried to retrieve the FPGA ID; if the device supports encryption, the bitstream can be encrypted. Next, the bitstream is transmitted to the device, and the device reconfigures itself using its built-in reconfiguration logic.

[0157] Having thus described several aspects of at least one embodiment of this invention, it is to be appreciated that various alterations, modifications, and improvements will readily occur to those skilled in the art.

[0158] Such alterations, modifications, and improvements are intended to be part of this disclosure, and are intended to be within the spirit and scope of the invention. Further, though advantages of the present invention are indicated, it should be appreciated that not every embodiment of the technology described herein will include every described advantage. Some embodiments may not implement any features described as advantageous herein and in some instances one or more of the described features may be implemented to achieve further embodiments. Accordingly, the foregoing description and drawings are by way of example only.

[0159] Various aspects of the present invention may be used alone, in combination, or in a variety of arrangements not specifically discussed in the embodiments described in the foregoing and is therefore not limited in its application to the details and arrangement of components set forth in the foregoing description or illustrated in the drawings. For example, aspects described in one embodiment may be combined in any manner with aspects described in other embodiments.

[0160] Also, the invention may be embodied as a method, of which an example has been provided. The acts performed as part of the method may be ordered in any suitable way. Accordingly, embodiments may be constructed in which acts are performed in an order different than illustrated,

which may include performing some acts simultaneously, even though shown as sequential acts in illustrative embodiments.

[0161] Such alterations, modifications, and improvements are intended to be part of this disclosure, and are intended to be within the spirit and scope of the invention. Further, though advantages of the present invention are indicated, it should be appreciated that not every embodiment of the invention will include every described advantage. Some embodiments may not implement any features described as advantageous herein and in some instances. Accordingly, the foregoing description and drawings are by way of example only.

[0162] All definitions, as defined and used herein, should be understood to control over dictionary definitions, definitions in documents incorporated by reference, and/or ordinary meanings of the defined terms.

[0163] The indefinite articles "a" and "an," as used herein in the specification and in the claims, unless clearly indicated to the contrary, should be understood to mean "at least one."

[0164] The phrase "and/or," as used herein in the specification and in the claims, should be understood to mean "either or both" of the elements so conjoined, i.e., elements that are conjunctively present in some cases and disjunctively present in other cases. Multiple elements listed with "and/or" should be construed in the same fashion, i.e., "one or more" of the elements so conjoined. Other elements may optionally be present other than the elements specifically identified by the "and/or" clause, whether related or unrelated to those elements specifically identified. Thus, as a non-limiting example, a reference to "A and/or B", when used in conjunction with open-ended language such as "comprising" can refer, in one embodiment, to A only (optionally including elements other than B); in another embodiment, to B only (optionally including elements other than A); in yet another embodiment, to both A and B (optionally including other elements); etc.

[0165] As used herein in the specification and in the claims, the phrase "at least one," in reference to a list of one or more elements, should be understood to mean at least one element selected from any one or more of the elements in the list of elements, but not necessarily including at least one of each and every element specifically listed within the list of elements and not excluding any combinations of elements in the list of elements. This definition also allows that elements may optionally be present other than the elements specifically identified within the list of elements to which the phrase "at least one" refers, whether related or unrelated to those elements specifically identified. Thus, as a non-limiting example, "at least one of A and B" (or, equivalently, "at least one of A or B," or, equivalently "at least one of A and/or B") can refer, in one embodiment, to at least one, optionally including more than one, A, with no B present (and optionally including elements other than B); in another embodiment, to at least one, optionally including more than one, B, with no A present (and optionally including elements other than A); in yet another embodiment, to at least one, optionally including more than one, A, and at least one, optionally including more than one, B (and optionally including other elements); etc.

[0166] Use of ordinal terms such as "first," "second," "third," etc., in the claims to modify a claim element does not by itself connote any priority, precedence, or order of one claim element over another or the temporal order in which acts of a method are performed, but are used merely as labels to distinguish one claim element having a certain name from another element having a same name (hut for use of the ordinal term) to distinguish the claim elements.

[0167] Also, the phraseology and terminology used herein is for the purpose of description and should not be regarded as limiting. The use of "including," "comprising," or "having," "containing," "involving," and variations thereof herein, is meant to encompass the items listed thereafter and equivalents thereof as well as additional items.

LIST OF REFERENCES

[0168] The following references are hereby incorporated by reference in their entireties:

[0169] [Ref. 1] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. FPGA-oriented Security. Introduction to Hardware Security and Trust/eds. M. Tehranipoor and C. Wang. Springer, pages 195-231, 2011.

[0170] [Ref. 2] Tim Guneysu et al. Dynamic intellectual property protection for reconfigurable devices. In ICFPT, pages 169-176. IEEE, 2007.

[0171] [Ref. 3] Ed Peterson. Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs. Technical report, Xilinx, 2011.

[0172] [Ref. 4] Altera. Protecting the FPGA design from common threats. Technical report, Altera, 2009.

[0173] [Ref. 5] Sergei Skorobogatov and Christopher Woods. Breakthrough silicon scanning discovers backdoor in militarychip. Springer, 2012.

[0174] [Ref. 6] Amir Moradi et al. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx Virtex-II FPGAs. In CCS, pages 111-124, 2011.

[0175] [Ref. 7] Siddika Berna O"rs et al. Power-analysis attacks on an FPGA—first experimental results. In CHES, pages 35-50. Springer, 2003.

[0176] [Ref. 8] Francois-Xavier Standaert et al. Power analysis attacks against FPGA implementations of the DES. In FPLA, pages 84-94. Springer, 2004.

[0177] [Ref. 9] E'ric Rannaud. From the bitstream to the netlist. In ACM/SIGDA symposium on Field programmable gate arrays, pages 264-264. ACM, 2008.

[0178] [Ref. 10] Robert McEvoy et al. Differential power analysis of HMAC based on SHA-2, and countermeasures. In Information security applications, pages 317-332. Springer, 2007.

[0179] [Ref. 11] P-Y Chen et al. Interconnection networks using shuffles. Computer, (12):55-64, 1981.

[0180] [Ref. 12] Ulrich Ruhrmair et al. PUF modeling attacks on simulated and silicon data. IEEE TIFS, 8(11): 1876-1891. 2013.

[0181] [Ref. 13] Aswin Raghav Krishna et al. MECCA: a robust low-overhead PUF using embedded memory array. In CHES, pages 407-420. 2011.

[0182] [Ref. 14] A. Vijayakumar and S. Kundu. A novel modeling attack resistant PUF design based on non-linear voltage transfer characteristics. In DATE, pages 653-658, March 2015.

[0183] [Ref. 15] IP Cores. UCore-Compact Advanced Encryption Standard (AES) Core. Online, 2006.

[0184] [Ref. 16] Panu H"am"al"ainen et al. Design and implementation of low-area and low-power AES encryption hardware core. In DSD (EUROMICRO), pages 577-583. IEEE, 2006.

[0185] [Ref. 17] Helion. AES Cores. Online. 2014.

[0186] [Ref. 18] CAST. AES-C: AES Optimized Encryption/Decryption Core. Online.

[0187] [Ref. 19] R. K. Soni, "Open Source Bitstream Generation for FPGAs (Doctoral dissertation, Virginia Tech), 2013.

What is claimed is:

1. A programmable device, comprising:
an external interface;
a first circuit configured to generate an identifier;
a second circuit configured to transmit through the external interface at least one response to one or more messages received through the external interface, wherein at least a portion of the at least one response is based at least in part on the identifier;
a third circuit configured to perform a de-obfuscating function on a bitstream, wherein the de-obfuscating function is based at least in part on the identifier.

2. The programmable device of claim **1**, wherein the programmable device is a field programmable gate array (FPGA).

3. The programmable device of claim **1**, wherein:
at least a portion of the identifier is based on a plurality of selectively blown fuses in the programmable device.

4. The programmable device of claim **1**, wherein:
at least a portion of the identifier has a value that varies over time.

5. The programmable device of claim **1**, wherein:
the third circuit comprises at least one sub-circuit configured to selectively permutate the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier.

6. The programmable device of claim **5**, wherein:
the third circuit comprises a plurality of sub-circuits, connected in series, wherein each of the plurality of sub-circuits is configured to selectively permutate the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier.

7. A method of securely programming a programmable device, the method comprising:
obtaining an identifier from the programmable device;
obfuscating a bitstream based at least in part on the identifier; and
sending the obfuscated bitstream to the programmable device.

8. The method of claim **7**, wherein obtaining the identifier comprises:
sending a sequence of challenges to the programmable device;
receiving a sequence of responses to the sequence of challenges from the programmable device; and
determining, based on the sequence of responses, the identifier for the programmable device.

9. The method of claim **7**, further comprising:
authenticating the programmable device based on the identifier in relation with an authorized identifier list.

10. The method of claim **9**, wherein authenticating the programmable device based on the identifier in relation with an authorized identifier list comprises:
obtaining the authorized identifier list from an external source.

11. The method of claim **10**, wherein obtaining the authorized identifier list from an external source comprises:
communicating with the external source using secure communications.

12. The method of claim **7**, wherein obfuscating the bitstream comprises:
permutating the bitstream.

13. The method of claim **7**, wherein obfuscating the bitstream comprises:
iteratively permutating the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the identifier.

14. The method of claim **7**, wherein obfuscating the bitstream further comprises:
generating a key based on the identifier;
obfuscating the bitstream by performing a plurality of obfuscation functions, each of the plurality of obfuscation functions being based on the key.

15. The method of claim **14**, wherein performing a plurality of obfuscation functions comprises:
iteratively permutating the bitstream such that a position within the bitstream of at least a portion of the bitstream is changed based at least in part on the key.

16. The method of claim **7**, wherein obfuscating the bitstream based on the at least one identifier comprises:
applying a plurality of permutation levels, the plurality of permutation levels further comprising a first level, a second level and a third level, wherein:
the first level comprises permutation of portions of the bitstream that specify an input ordering of a look up table (LUT);
the second level comprises permutation of the portion of the bitstream that specifies a content of the LUT;
the third level comprises a block based permutation of the entire bitstream.

17. A method of securely operating a programmable device that receives a programming bitstream, the method comprising:
generating a pseudo-random identifier;
transmitting a sequence of responses based on the identifier in response to receiving a sequence of challenges, wherein at least a portion of the sequence of responses is based at least in part on the identifier;
de-obfuscating a received bitstream based on the identifier; and
programming programmable circuitry within the programmable device based on the de-obfuscated bitstream.

18. The method of claim **17**, wherein de-obfuscating the bitstream based on the identifier comprises:
permutating the bitstream based on the identifier.

19. The method of claim **17**, wherein de-obfuscating the bitstream based on the identifier comprises:
transforming the bitstream based on a plurality of fuses in the programmable device that are selectively blown.

20. The method of claim **17**, wherein de-obfuscating the bitstream based on the identifier comprises:
applying a plurality of permutation levels, the plurality of permutation levels further comprising a first de-obfus-

cation level, a second de-obfuscation level and a third de-obfuscation level, wherein:

the first de-obfuscation level comprises permutating the bitstream on a first portion of the programmable device;

the second de-obfuscation level comprises permutating the bitstream on a second portion of the programmable device;

the third de-obfuscation level comprises permutating the bitstream on a third portion of the programmable device.

* * * * *