This paper proposes a technique based on Physical Unclonable Functions (PUFs), obfuscation, and Dynamic Partial Self Reconfiguration (DPSR) to protect partial FPGA configuration bitstreams from cloning and reverse engineering. With the aid of this technique, we are able to do the equivalent of partial bitstream encryption on low-cost FPGAs, which is only featured on high-end FPGAs. Low-cost FPGAs do not even have built-in support for encrypted (full) bitstreams. Through DPSR, our PUF implementation does not steal real estate from the encrypted design. We also present a new DPSR flow for Xilinx FPGAs, which is difference-based but still allows modular design. It works regardless of the amount of difference between Partial Reconfiguration (PR) modules and is called DPSR-LD, where LD stands for Large-Difference. DPSR-LD is an enabler especially for Spartan-6 FPGA family, as Xilinx currently supports PR on Spartan-6 only through the difference-based flow and only for small differences. Our DPSR-LD also includes a controller that interfaces to the ICAP and can process compressed bitstreams. It is called ICAP+ and occupies only 1% of Spartan-6 slices. © 2012 Elsevier Ltd. All rights reserved.
products (also available on low-end Spartan-6 FPGAs). Device DNA is a per-chip ID code that is factory-set via e-fuses (partly user-set on high-end FPGAs). One can use a DNA-based authentication to at least prevent cloning. If the design compares the device’s DNA with the DNA value embedded in the bitstream, then the same bitstream would not function on a different device as the DNA value in the copied bitstream would not match the new device’s DNA. The key inside the bitstream can be a scrambled version of the DNA. However, designs with Device DNA can still be cloned. One can read Device DNA from the device, reverse engineer bitstream into a netlist [1] (by comparing bitstreams from multiple FPGAs), locate where the Device DNA connects to the design, disconnect DNA port and hard-wire it to the DNA value of the cloned device. In addition, Device DNA can be classified as part of reproducible nonfunctional identification category of the IC metering taxonomy in [3]. Since the burned fuses are reproducible at the foundry, Device DNA cannot be used as a countermeasure for a foundry piracy attack.

As mentioned earlier, the flexibility of FPGAs allows them to be customized for various applications. Even individual applications can benefit from reconﬁgurability during runtime to perform different tasks at different times. If not all of an application’s conﬁgurations ﬁt on the hardware simultaneously, they can be swapped in and out as needed. For example, a cellular phone switches from a GSM to CDMA network or a TV switches from 3D video feed to 2D. Through Dynamic Partial Self Reconﬁguration (DPSR), an already running FPGA decides what subdesigns to download from Flash memory (i.e., new features) and reclaims resources taken up by subdesigns not needed anymore (i.e., old features). DPSR saves hardware resources as well as reconﬁguration time [5] if the conﬁgurations are smaller than the full device. Although the DPSR concept has been around for more than a decade, it is still not widespread in the industry. The main reason for this is the lack of good commercial design tools in support of DPSR. Another reason is that these tools support only high-end devices. Xilinx offers FPGAs with high logic density targeting high-performance applications. Compared to Xilinx Virtex-5 and Virtex-6 families, the cheaper Spartan-6 FPGAs provide a lower logic density and consume less power. However, there is no DPSR tool support for the cheaper and low-power Spartan-6, while there are tools available for Virtex-5 and Virtex-6.

To address both bitstream protection and DPSR, we need partial bitstream encryption. That is only supported on high-end FPGAs such as Virtex-5 and Virtex-6. For example, Spartan-6 family, which mostly consists of low-end FPGAs, does not support any form of encryption (even for full bitstreams) except for its most high-end FPGAs. In addition to the lack of DPSR tool support, there is also no such tool support for partial bitstream encryption of the low-end Spartan-6.

In this paper, in order to achieve partial bitstream protection for low-end FPGAs:

1. We propose to use Physical Unclonable Functions (PUFs) [6–10] in place of reproducible nonfunctional IDs such as Device DNA [4]. PUFs offer intrinsic fingerprints for an FPGA chip by exploiting the uniqueness of slight manufacturing variations that occur between chips. In other words, each chip is uniquely and unclonably identified by a PUF module. This class of IC identiﬁcation is called unclonable identiﬁcation according to the IC metering taxonomy in [3].

2. We propose to use PUF key-based active obfuscation [11–13] in place of encryption. PUF key-based active obfuscation is the process of blending the chip’s signature obtained from a PUF module with the IP’s functionality. When several chips from the same family and type are configured with the same obfuscated bitstream, only the dedicated chip runs correctly, while the rest of the chips malfunction. We accomplish obfuscation at Register Transfer Level (RTL), where we do not only make the design hard to trace but also the design does not work as expected unless the correct PUF key is applied. The design is injected extra gates that turn into wires with the correct signature. A stable PUF requires quite a few bits of PUF and may not ﬁt in the FPGA together with the design. We also use a Time-Division-Multiplexed-PUF (TDM-PUF) [14] which divides a single and long PUF into smaller PUFs that run at different time segments.

3. We propose a difference-based DPSR ﬂow, which we call DPSR-LD, where LD stands for Large-Difference, in place of Xilinx PR tool. This ﬂow works regardless of the amount of difference between Partial Reconﬁguration (PR) modules. Our proposed ﬂow enables PR especially on low-end devices such as Spartan-6 family, as Xilinx currently supports PR on Spartan-6 only through the difference-based PR (diffPR) ﬂow and only for small differences. Xilinx’s diffPR ﬂow [15] suggests using diffPR when making small on-the-fly changes to design parameters such as logic equations, ﬁlter parameters, and I/O standards. Xilinx does not recommend diffPR ﬂow [15] for large changes.

The organization of this paper is as follows. Section 2 gives the background and our methodology on unclonable identiﬁcation, PUF key-based obfuscation, and our novel DPSR ﬂow (DPSR-LD). Section 3 presents implementation and results by a proof-of-concept GPIO controller on Spartan-6 FPGA. Finally, Section 4 concludes the paper.

2. Background and methodology

This paper proposes a partial bitstream protection technique based on unclonable identiﬁcation, PUF key-based active obfuscation, and a novel difference-based DPSR ﬂow (DPSR-LD). In the following subsections, we will explain each of the three components of our technique in detail.

2.1. Unclonable identiﬁcation

PUFs are considered unclonable, as they depend on random parameters in the manufacturing process. PUFs can be used to identify a device and are an alternative to embedded/printed serial numbers (e.g., Xilinx Device DNA). PUFs consist of
When we add a large number of new states a number of added states and transitions. If the original FSM has showed how to devise an obfuscated FSM that is provably secure. The obfuscated FSM includes the original FSM, along with controller FSM of the circuit, which controls the functional modes based on application of the PUF response. Koushanfar demonstrated proofs for developing secure integrated circuit (IC) control mechanism with the functional description of the design as well as unique and unclonable IC identifiers. In an earlier work, we combined PUF key-based obfuscation and multi-boot feature of Spartan-6 devices to achieve full bitstream protection. In multi-boot, FPGA has to overwrite its configuration completely and externally from a Flash memory. In this paper, we extend our combined PUF key-based obfuscation and multi-boot feature of Spartan-6 devices to achieve full bitstream protection. In multi-boot, FPGA has to overwrite its configuration completely and externally from a Flash memory. In this paper, we extend our

2.1. Our customization of Anderson's PUF for Spartan-6

Since Anderson's PUF's original implementation was on Virtex-5, we modified Anderson's PUF for Spartan-6 (Fig. 1). As opposed to Virtex-5 devices, Spartan-6 flip-flops cannot be initialized to an initial state independent of set/reset state. To overcome this, a simple modification was done on the original Anderson's PUF design to make it work on Spartan-6. In Fig. 1, an extra signal (shown as PUF reset) is used to initialize the flip-flops, and it is connected to the clock enable port of the flip-flop. The output of the carry chain is connected to the reset port, and whenever a glitch is produced, it is captured. Note that reset has a higher priority over the clock enable signal. Anderson's PUF cell utilizes two vertical neighboring slices (typically SLICEMs – 25% of Spartan-6 slices are SLICEMs). Our target Spartan-6 (XC6SLX45 on a Digilent Atlys board) has 6,822 slices. If the design was implemented with a pure static flow, a 64-bit PUF key would require 128 SLICEMs (64 × 2). Since our flow is based on DPSR, only 64 SLICEMs are used (=3.75% of Spartan-6 area) where we implement 2 PUF farms (Fig. 2) each with 32-bits.

2.2. PUF key-based active obfuscation

Obfuscation is intentionally modifying the description or structure of a circuit in order to conceal its functionality and is making it significantly more difficult to reverse-engineer. It is also different from cryptography-based protection techniques common in FPGA design flow. There are two approaches for hardware obfuscation. The first one is called passive obfuscation, in which the comprehensibility of the description of the hardware (Hardware Description Language (HDL) code) is concealed without changing the functionality. As opposed to passive obfuscation, active obfuscation directly alters the functionality of the circuit. Active obfuscation can be done either at gate-level or RTL level. Active obfuscation techniques are often “key-based” in which normal functionality of the obfuscated design can only be unlocked by applying a single pre-determined key or a sequence of secret keys at the input; otherwise the circuit operates incorrectly.

Obfuscation received skepticism in the past. Barak et al. proved the existence of certain classes of functions that cannot be obfuscated. However since then, several works have shown the feasibility of secure “key-based” obfuscation. Recently, Koushanfar demonstrated proofs for developing secure integrated circuit (IC) control mechanism with the functional description of the design as well as unique and unclonable IC identifiers. In an earlier work, we combined PUF key-based obfuscation and multi-boot feature of Spartan-6 devices to achieve full bitstream protection. In multi-boot, FPGA has to overwrite its configuration completely and externally from a Flash memory. In this paper, we extend our earlier work to achieve partial bitstream protection by replacing multi-boot with DPSR-LD.

PUF key-based active obfuscation is done by embedding a well-hidden finite state machine (FSM) or modifying the controller FSM of the circuit, which controls the functional modes based on application of the PUF response. Koushanfar showed how to devise an obfuscated FSM that is provably secure. The obfuscated FSM includes the original FSM, along with a number of added states and transitions. If the original FSM has |S| states, it can be implemented using \( \log(|S|) \) flip-flops (FFs). When we add a large number of new states |S'|, |S| + |S'| states can be implemented by a linear growth in the number of FFs that is \( \log(|S| + |S'|) \). Upon-power-up, the initial values of the design's added FFs are determined by the unique response from the PUF module. The number of added FFs should be large enough so that there is a high probability that PUF response sets the initial power-up state to one of the added states. Then one needs to provide a sequence of pass-keys (shown as (PUF, K1, PUF, K2, ..., PUF, KN) in Fig. 3) required for traversal from the power-up state to the reset state of the original FSM. Fig. 3 depicts an obfuscated FSM in which three state spaces are shown: (i) original state space, (ii) initialization state space, (iii) added state space.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Silicon PUF comparison.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement and routing complexity</td>
<td>Manual routing required</td>
</tr>
<tr>
<td>Ring oscillator PUF</td>
<td>High</td>
</tr>
<tr>
<td>Arbiter PUF</td>
<td>High</td>
</tr>
<tr>
<td>Anderson PUF</td>
<td>Low</td>
</tr>
</tbody>
</table>
and (iii) isolation state space. Isolation and initialization state spaces consist of new states. Depending on the PUF response, the power-up state can be in either isolation or initialization state space. In Fig. 3 the power-up state is placed in the initialization state space. After application of a sequence of pass-keys (that is also made up from PUF response), an initialization process sets the next state to the reset state of the original state space. During initialization, application of even a single wrong key sets the next state to a state in isolation state space. From the attacker’s point of view, the chance of finding

Fig. 1. Anderson PUF implementation on Spartan-6.

Fig. 2. PUF farm.
out the correct PUF key to unlock the obfuscated design is pretty low, which is \( \frac{1}{(3 + 3)^{2}} \), where \( N \) denotes the number of transitions in the initialization sequence and \( M \) denotes the number of bits in the pass-key (PUF\(_{K1}\), PUF\(_{K2}\), ..., PUF\(_{Kn}\)).

Provably secure obfuscation has an overhead in resource utilization and extensive experimental results were reported in [13] on the ISCAS sequential benchmark suite. Their results indicated large fluctuations among the circuits on area overhead, for large circuits the overhead on the average was given as 13%. In addition to area overhead due to obfuscation, PUFs also require considerable amount of area. In [13], the area overhead for the PUF modules were not included in the results and no solution to this problem was given in [13]. However, we use DPSR to solve this problem where PUFs are placed in dynamic regions which are then reclaimed for the actual (protected) design. To the best of our knowledge, our earlier paper [14] and this paper are the first papers in the literature which propose DPSR to remove the area overhead of PUF modules in PUF key-based active obfuscation using FPGA design flow. In the next subsection, we explain PR concepts in detail and our novel difference-based DPSR-LD flow.

### 2.3. Partial reconfiguration

While the FPGA configuration method offers reprogramming flexibility in its nature, PR takes it one step further. PR is the ability for a portion of an FPGA to be reprogrammed while the remainder of the system stays unchanged. If we look at PR from an operational point of view, there can be two cases: Dynamic PR (DPR) and Static PR (SPR). DPR is also known as active PR. DPR allows device reconfiguration during runtime while rest of the device is still functioning. On the other hand, in SPR the FPGA is not active during reconfiguration, that is while the partial bitstream is loading on the FPGA, the device is stopped and restarted after the end of configuration. SPR is an obsolete technique, which is replaced by DPR. DPR allows creating efficient systems with FPGAs, where devices operate in a mission critical environment that cannot be disrupted while some subsystems are being redefined. DPR can be driven either from outside the FPGA with a processor chip or a full-blown computer. It can also be driven from inside the FPGA (i.e., DPSR) with a processor core or a dedicated core.

Although DPR is a great concept, in practice it has difficulties and some downsides. In order to create a system with DPR, there are several additional steps in both design and implementation phases. If the reconfigurable part of the system is a complex component, the designer must manage certain implementation steps specifically for PR implementation. In other words, the system should be partitioned into its static and time-varying (dynamic) parts and then development should be done based on the requirements of these partitions. The dependencies between dynamic modules connected to the same system point should be eliminated since those parts will never exist in the system at the same time interval. On the other hand, there are several physical constraints due to the FPGA platform and project design requirements. In implementation phase, the physical placements of the static and dynamic modules are directed to specified parts of FPGA layout. The physical layout of the system should be designed to provide resources to both static and dynamic parts adequately. Additional reset and handshaking circuitry is necessary in most cases. To handle these requirements, Xilinx offers two different PR implementation methodologies:

1. **DiffPR** [15] is offered for making small changes in FPGA functionality. The PR bitstream is produced by BitGen utility (part of Xilinx’s free ISE WebPACK suite) with \(-r\) option on the bitstream of current design and new netlist (ncd file).
2. **Module-based PR** [19] is more general-purpose. The designer specifies portions of the device that can be reconfigured, which we call “dynamic regions” (and the designs that fill them are “PR modules”). Design can be done in parallel for all PR modules and static module. Module-based PR is only supported on Virtex families through PlanAhead tool (needs a special paid license) and was previously supported through Early Access Partial Reconfiguration [19] tools.
DiffPR is normally meant for small changes, especially on-the-fly modification of parameters kept in LUTs, BRAMs, IO pads. The design differences in DiffPR can be specified at RTL level. However, a small RTL change sometimes results in a large netlist change and can cause a failure. Also, even when a diffPR design properly functions, DPSR may not work as the PR bitstream may reprogram areas occupied by the static logic – due to changes in routing. The safest path for DiffPR is using Xilinx FPGA Editor [19], which lets the designer make local surgery in the mapped and placed & routed (PAR’ed) design such as modifying LUT equations, BRAM contents/modes, I/O pin parameters, flip-flop initialization and reset values, pull-ups or pull-downs on external pins. Bitstreams produced based on .ncd files modified by FPGA Editor are guaranteed to have small changes as long as we make only a few changes in FPGA Editor. Even though it is possible to change routing in FPGA Editor, it is not recommended due to the possibility of internal contention during reconfiguration. If routing changes are desired, diffPR flow described in [15] is not recommended. Making large modifications using diffPR leaves artifacts from the previous design. Even identical (static) parts of the design contain differences after synthesis and implementation steps.

Here are some reasons why DiffPR can cause failures:

- Partial and static modules are synthesized and implemented together (unlike module-based PR).
- Static parts, which are synthesized alongside with different PR modules, may be synthesized differently. Two identical static designs with different PR modules become different after synthesis and implementation steps.

2.3.1. Our DPSR-LD flow

In this subsection, we introduce our DPSR-LD flow. Although our flow makes use of BitGen utility (part of Xilinx’s free ISE WebPACK suite) with –r option, it works for large differences as well and starts by defining dynamic region just like module-based PR. Our DPSR-LD flow makes use of FPGA Editor utility (part of Xilinx ISE Design Suite) to remove the artifacts that can be left from the previous design in the dynamic region and solves several problems listed above. Before we introduce our DPSR-LD solution for the low-cost Xilinx FPGAs (such as Spartan-6), below we first introduce the terminology.

The static region of an FPGA design is an area that is configured only once during runtime. Dynamic region is an area that is reconfigured multiple times. A PR module configures only a dedicated dynamic region. A complete design configuration is a set of bitstreams that fully configures the FPGA, i.e., static and dynamic regions. Suppose an FPGA has one dynamic and one static region, and the dynamic region can be reconfigured for \( n \) different partial designs. Consider, for example, a complete design configuration \( F_i \) composed of a static design, \( S \) and \( i \)th PR design, \( D_i \). Following are the steps of DPSR-LD that brings \( F_1 \rightarrow F_2 \) and then to \( F_3 \) and so on up to \( F_n \):

1. For all \( i \), generate the complete design, \( F_i \) (with dynamic region constraints) that puts together \( S \) and \( D_i \).
2. Take \( F_1 \) and copy its PAR’ed native circuit description (.ncd file), then remove the logic in all dynamic regions using a batch script that runs FPGA Editor. Call the resulting design \( F_B \), where B means Blank.
3. For all \( i \) except \( n \), take the difference, \( F_B \) minus \( F_i \). Call the resulting partial bitstream P_itoB (Fig. 4).
4. For all \( i \) except 1, take the difference, \( F_i \) minus \( F_B \). Call the resulting partial bitstream P_Bto_i (Fig. 4).
5. Our configuration controller (ICAP+) interfaces to the Internal Communication Access Port (ICAP) and reconfigures the FPGA in the following order: \( F_1, P_1\_\text{itoB}, P_2\_\text{toB}, \ldots, P_n\_\text{itoB}, P_n\_\text{Bto} \) (Fig. 4).

In step 1, the dynamic region constraints are determined and entered as implementation constraints [19], which guide FPGA implementation tools for mapping, PAR, timing. They are usually placed in the User Constraints File (UCF), but may also exist in the HDL code or in a synthesis constraints file. After constraints creation, \( n \) different complete design configurations are implemented by running the Xilinx tool flow. Xilinx tool flow consists of (1) synthesis, (2) translation into a native generic database, (3) mapping the logic to the components (logic cells, I/O cells, etc.) in the target Xilinx FPGA, (4) PAR, (5)
In step 2, logic in dynamic regions of the first complete design configuration’s NCD \((F_1.ncd)\) is removed by the help of a batch script we wrote for FPGA Editor. Components that belong to dynamic region are selected and deleted by commands select and delete. FPGA Editor is a program for displaying and configuring layout of Xilinx FPGAs. It requires an NCD file, which contains the logic components in the design (e.g., CLBs and IOBs). After select and delete operations, the changes are saved in another NCD file. To prepare an already PAR’ed dynamic region for a new PR module, all components in the dynamic region (LUTs, flip flops, etc.), and nets connected to the anchor points (bridge between static and dynamic regions) should be removed from the netlist. FPGA Editor command line “select dynamic_region_name” is used to clean the components in a PR module. The nets at anchor points can be addressed and cleaned through signal names in the top-level RTL as they still keep their signal names in the NCD. After removing the components and nets, the dynamic region becomes truly blank, and we obtain \(F_0.ncd\) and \(F_0.bit\).

In step 3, \((n - 1)\) partial bitstreams that blank out the dynamic regions \((P_{1toB}.bit)\) are generated by BitGen with the \(-r\) switch and \(F_0.ncd\), \(F_i.bit\) as arguments in each run.

Step 4 generates \((n - 1)\) partial bitstreams \((P_{Bto1}.bit)\) that program the dynamic regions to create a complete design (done by BitGen \(-r\) and \(F_0.ncd\), \(F_0.bit\) as arguments).

In step 5, our controller interfacing to ICAP (called ICAP+) applies the desired configuration order. ICAP is the internal communication access port that provides configuration access to the FPGA logic. Partial bitstreams are stored in BRAM, ICAP + reads and decompresses them, then reconfigures the dynamic regions. Configuration sequence is \(F_1\), \(P_{1toB}\), \(P_{Bto2}\), \(P_{2toB}\), \ldots, \(P_{Bto}\) as shown in Fig. 5.

In our flow, static parts of both \(F_i\)’s and \(F_0\) are guaranteed to be implemented identically. Since we create the difference by deleting dynamic modules, all problems are eliminated. The resulting partial bitfiles (i.e., bitstreams) are both region and pin compatible by the help of heavy constraining of the PAR steps.

**2.2.1.1. Our ICAP controller and bitstream compression.** FPGA configuration memory is volatile and needs to be reprogrammed at every system power up. The process of loading the bitstream is also called configuration. FPGAs can be (re)configured through Boundary-Scan (JTAG), Select-MAP, Serial Peripheral Interface (SPI), and ICAP. JTAG, Select-MAP, and ICAP could be used for PR. PR is not achievable [20] through JTAG port of Spartan-6, ICAP is chosen for the configuration interface. Our ICAP controller, namely ICAP+ (Fig. 6), consists of a BRAM module to store compressed partial bitstreams, a module to decompress the bitstreams, and a top-level module to send uncompressed data from BRAM to the ICAP interface upon receiving the select signal for which PR module to reconfigure. To reduce configuration time, bitstream file is compressed offline and is decompressed on-the-fly.

Regarding bitstream compression, Li et al. [21] investigated the redundancy in various bitstream files and applied compression algorithms including Huffman coding, Arithmetic coding, and LZ compression. Their simulation results indicated that a compression ratio of 4:1 can be achieved. Dandalis et al. [22] proposed a dictionary-based compression approach. They showed 11–41% savings in memory for configuration bitstreams of several applications. Pan et al. [23] proposed intra-bitstream compression technique for Xilinx Virtex family as opposed to inter-bitstream compression techniques. They reported that their approach achieved 27–76% improvement over the DV and LZSS algorithms.

Most proposed bitstream file compression techniques are based on complicated compression algorithms in order to achieve high compression ratios. Koch et al. [24] investigated several compression algorithms with respect to the achievable compression ratio, throughput, and hardware overhead. They reported that Huffman encoding enhances the compression ratio to almost 40% and a combined LZSS and Huffman encoding to 34%. However, to achieve such compression ratios, significant chip area must be allocated for the decompressor. In [24] it is also reported that about 5000 LUTs are required for the combined LZSS-Huffman accelerator whereas less than 100 LUTs are needed for run-length encoding (RLC) and LZSS hardware decompressors with 50% compression ratios.

Liu et al. [25] proposed a simple compression/decompression technique in which a code word including the count of the repeated words is entered after the repeated words for Xilinx Virtex-4 FPGA. Since we mostly target low-cost Xilinx Spartan-6 FPGAs, we tailored compression to Spartan-6. Spartan-6 configuration word size is 16 bits. Spartan-6 bitstreams have

![Fig. 5. DPSR-LD configuration sequence.](image)
redundancy in the form of many NULL (0000) or NOOP (FFFF) words (usually in a repeated fashion). This redundancy can be exploited through RLC. Our bitstream compressor always compares the incoming word with the previous. Our compressed bitstream has 17-bit words. The first bit is a flag. If the flag is 1, the next 16 bits is a value passed as is from the input bitstream. If the flag is 0, then the next 16 bits is the number of times the previous 16-bit word is repeated. We usually get over 2X compression using this simple approach. The configuration time \( t_{cfg} \) is the product of the size of the partial bitstream in words with the configuration clock period \( T_{cfg} \). For example, the configuration time needed to configure a partial bitstream of 12,567 16-bit words through ICAP is 0.252 ms using a 20 ns clock period. The configuration to blank out this dynamic region is also 0.252 ms. ICAP+ occupies about 1% of the total number of slices (83 slices out of 6822).

2.3.1.2. User constraints. DPSR-LD relies on the UCF for properly generating partial bitstreams. We use implementation, floor-planning, and placement constraints such as LOC and AREA_GROUP constraints to enable partitioning of the design into physical regions for mapping, packing, placement, and routing. We use GROUP = CLOSED setting in the area group constraints so as not to allow logic outside the area group to be combined with logic inside the area group. We use PLACE = CLOSED setting in order not to allow components (logic cells, I/O cells, etc.) that are not members of the area group to be placed within the RANGE defined for the area group where RANGE specifies the area with slice coordinates.

Bus macros were required in module-level PR schemes such as Xilinx Early Access PR (EAPR) flow [19]. Bus macros provide a means of locking the routing between PR modules and the static design, making PR modules pin-compatible with each other so that they can be swapped. In EAPR flow, all connections between the dynamic parts of design and static design must go through a bus macro with the exception of global signals (BUFG global clocks, GND, VCC), which are handled automatically by the tools. Bus macros are provided with EAPR in the form of pre-placed, pre-routed hard macros. Bus macro instantiations are no longer required in the latest Xilinx PR tool [19]. However, neither the latest Xilinx PR tool nor EAPR supports Spartan devices.

Our DPSR-LD does not require hard bus macros. Instead, we place a simple block that serves as a bridge between static and dynamic areas. These blocks are simple FFs in FPGA slices with their enable controlled by static modules. This bridge can both capture the output of a PR module and drive its inputs as well. FPGA slices have multiple FFs, and hence, a single FPGA slice can make many connections between static and dynamic regions. These blocks are created through RTL coding and are instantiated in the top-level and are part of the static design. During synthesis, optimization between module boundaries is disabled. We fix the position of these modules by LOC and BEL constraints in the UCF. We guarantee mapping of the FFs to the same slice with LOC constraints and arranging them in the same order with BEL constraint. Xilinx PAR tool uses any routing resources available near the slice it is processing. Thus sometimes, neighboring slices’ switch box resources are used in order to complete routing (even if the logic placement for a region is locked). There is a great risk that a net from the static region could cross over the boundaries of a dynamic region, and this net could be corrupted. In order to avoid such situation, the boundaries of the dynamic region are surrounded by “placement prohibited” slices like the walls of a fortress. We use CONFIG PROHIBIT constraint to separate dynamic and static regions. By combining all of the above constraints, we guarantee that PR modules connect to the ports of FFs that serve as the bridges.

3. Implementation and results

Our first experiment is to measure resource overhead due to obfuscation. We developed a PERL script that automatically generates the HDL code of the obfuscated FSM for synthetic FSM benchmarks. Both the original and obfuscated FSMs are
synthesized and implemented by targeting Xilinx Spartan-6 XC6SLX45 FPGA with Xilinx ISE DS 13.2. Table 2 presents the resource utilization for the obfuscated as well as the original FSM benchmarks when obfuscation structure shown in Fig. 3 is employed with 32-bit state registers. As shown in Table 2, the number of the occupied slices for the obfuscated FSM is about five times the original.

Our second experiment is to measure the Hamming distances of the 64-bit PUF keys. We have implemented 2 PUF farms to generate a 64-bit key (32-bit from each) as well as a PUF data extractor with UART controller. We targeted five Digilent Spartan-6 (XC6SLX45) Atlys boards. We used the same PUF farms and PUF data extractor on each board. We extracted a 64-bit key from each of the five Atlys boards. Table 3 presents the hamming distances of the keys which are compared in pairs. If logic-0 and logic-1 were equally probable, one would expect the distribution to be clustered around an expected value of 32 bits. The average distance between any two pairs is 29.2 in Table 3, which is relatively close to the expected value.

Finally, we have applied our proposed partial bitstream protection technique on a General Purpose Input Output (GPIO) controller design. The design has 16 GPIO ports (with configurable directions). Operations on the ports are controlled by an FSM. We communicate with this FSM through commands (read, write, toggle, configure direction) over an RS-232 interface.

Original GPIO FSM has 23 states and at least 5 FFs are required to implement it \([|S|] = 23\) and \(K = \log\ (|S|)\) FFs. We applied obfuscation on the HDL of the GPIO design using the 64-bit PUF key \((\text{PUF}_{K63,0})\) of Board#1 based on FSM obfuscation shown as in Fig. 3. In Table 4, we give the resource utilization overhead after the obfuscation process. As seen in Table 4, we added 32 more FFs to the GPIO design and implemented an obfuscated FSM based on Fig. 3 of Section 2.2, where we encoded the original, initialization, and isolation state spaces as 32-bits. Obfuscated FSM has 23 states in the original, four states in the initialization, and \((2^{32} - 27)\) in the isolation state spaces. We initialized the FSM by setting the first 32 bits of the 64-bit PUF key \((\text{PUF}_{K63,32})\) to the power-up state. Note that the state encodings were tailored in such a way that the power-up state could only be either the dedicated start state of the initialization state space or one of the isolation states. We used the rest of the PUF bits \((\text{PUF}_{K31,0})\) as inputs in an initialization sequence of four cycles: (1) \(\text{PUF}_{K31,24}\), (2) \(\text{PUF}_{K3,16}\), (3) \(\text{PUF}_{K15,8}\), and (4) \(\text{PUF}_{K7,0}\). The resource utilization and obfuscation overhead are given in Table 4.

After we completed the PUF key generation and obfuscation, next we prepared the bitstreams using the DPSR-LD flow. We had one static and one dynamic region allocated in the target Spartan-6. The static part consisted of the top-level controller, UART controller, and ICAP+. Three designs (1) PUF1, (2) PUF2, and (3) obfuscated GPIO were targeted for the dynamic region. We generated one complete and three partial configuration bitstreams using DPSR-LD flow. In Figs. 7 and 8, configuration B configures the whole FPGA with static module and PUF1, configuration A blanks dynamic region, configuration C reconfigures dynamic region with PUF2, and configuration D reconfigures it with the obfuscated GPIO design. Note that configurations A, C, D are partial, only B is full. They are compressed (occupy 14% of BRAMs) and stored in BRAM for ICAP+ to decompress them and reconfigure on-the-fly. The boot-up sequence is as follows (Fig. 8):

1. B: 32-bit PUF1 signature is generated and written on registers.
2. A: dynamic region is cleared.
3. C: 32-bit PUF2 signature is generated and written on registers.
4. A: dynamic region is cleared.
5. D: GPIO block runs either in functional or nonfunctional state depending on the PUF signatures in registers.

In Table 5, we present our measured configuration times and bitstream file sizes (both uncompressed (B, A, C, D) and compressed (A, C, and D)) for four configurations used in the boot-up sequence. Note that the configuration B is a full bitstream and we used 1-bit SPI at 22 MHz to configure the chip, whereas we use 16-bit ICAP at 20 MHz to configure the chip with partial configurations (A, C, and D). Partial bitstream configuration times are noticeably smaller than the full bitstream configuration times. This is mainly because of the differences in file sizes and configuration interfaces (1-bit SPI versus 16-bit ICAP). In Table 5, the compressed partial bitstream file sizes are also given. A total of 24 KB BRAM is required to store them which is about 14% of the total BRAM resources of Spartan-6 (XC6SLX45) and is totally reusable after the boot-up sequence. In Fig. 9, two FPGA Editor screenshots are presented. The left image in Fig. 9 shows configuration B including PUF1, UART, and ICAP+ modules, whereas the right image shows the same area including UART and ICAP+ modules after configuration A is loaded. We applied the boot-up sequence in Fig. 8 on each of the five Atlys boards using the same bitstreams. As we expected, GPIO design ran correctly only on Board#1, not on other four Atlys boards.

### Table 2

| # Of states \(|S|\) | # Of inputs | # Of outputs | # Of occupied slices | Resource Overhead over Total \(\times 100\) (%) |
|------------------|-------------|--------------|----------------------|-----------------------------------------------|
|                  | Original    | Obfuscated   |                      |                                               |
| 8                | 32          | 32           | 19                   | 96                                            | 1.13                          |
| 16               | 32          | 32           | 24                   | 114                                           | 1.32                          |
| 32               | 32          | 32           | 39                   | 181                                           | 2.08                          |
| 64               | 32          | 32           | 79                   | 324                                           | 3.59                          |
| 128              | 32          | 32           | 144                  | 499                                           | 5.20                          |
Table 3
Hamming distances of PUF keys.

<table>
<thead>
<tr>
<th></th>
<th>Board #2</th>
<th>Board #3</th>
<th>Board #4</th>
<th>Board #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board #1</td>
<td>28</td>
<td>31</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>Board #2</td>
<td>–</td>
<td>–</td>
<td>33</td>
<td>28</td>
</tr>
<tr>
<td>Board #3</td>
<td>–</td>
<td>–</td>
<td>24</td>
<td>39</td>
</tr>
<tr>
<td>Board #4</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 4
Resource utilization for original versus obfuscated GPIO design.

<table>
<thead>
<tr>
<th>Logic utilization</th>
<th>GPIO original (#)</th>
<th>GPIO obfuscated (#)</th>
<th>Spartan-6LX45 (Resource Overhead over Total) × 100 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSM states</td>
<td>23</td>
<td>(23 (Orig.) + 4 (Init.) + 2^{22–27} (Iso.))</td>
<td>–</td>
</tr>
<tr>
<td>LUTs</td>
<td>127</td>
<td>210</td>
<td>0.30</td>
</tr>
<tr>
<td>FFs</td>
<td>54</td>
<td>86</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 5
Measured Spartan-6 configuration times and bitstream file sizes.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration interface</td>
<td>1-bit SPI @ 22 MHz</td>
<td>16-bit ICAP @ 20 MHz</td>
<td>16-bit ICAP @ 20 MHz</td>
<td>16-bit ICAP @ 20 MHz</td>
</tr>
<tr>
<td>Bitstream file size (KB)</td>
<td>1450</td>
<td>Original 38</td>
<td>Compressed 1</td>
<td>Original 24</td>
</tr>
<tr>
<td>FFs (#)</td>
<td>452</td>
<td>0</td>
<td>40</td>
<td>86</td>
</tr>
<tr>
<td>LUTs (#)</td>
<td>605</td>
<td>0</td>
<td>92</td>
<td>210</td>
</tr>
<tr>
<td>Configuration Time (ms)</td>
<td>700</td>
<td>0.95</td>
<td>0.60</td>
<td>0.93</td>
</tr>
</tbody>
</table>
4. Conclusions

Xilinx does not offer any partial bitstream encryption or PR support for low-cost FPGAs. This paper proposes a partial bitstream protection technique for low-cost Xilinx FPGAs using PUF key-based active obfuscation and our novel DPSR-LD flow. To the best of our knowledge, we developed the first partial bitstream protection technique which combines PUF key-based active obfuscation and DPSR. In this paper, our first contribution is to enable Anderson’s PUF design to work on Xilinx Spartan-6 devices. We evaluated the modified PUF design by considering the logic-0 and logic-1 distributions in the PUF responses which were found to be equally probable. In addition, we developed a tool that automatically generates the Verilog code of the obfuscated FSM for a given FSM RTL code using provably-secure obfuscation technique. Resource overhead of the obfuscation process is also analyzed with synthetic FSM benchmarks. Our second contribution is DPSR-LD, which enables DPSR for low-cost FPGAs with no Xilinx PR support. DPSR-LD utilizes diffPR and diffPR is only applicable for small changes in successive configurations. We demonstrated that DPSR-LD can be successfully applied to the cases where there are significant design changes between successive configurations. In DPSR-LD, FPGA Editor is used to remove the logic inside the dynamic regions which are determined by the implementation constraints. In addition, DPSR-LD does not require any hard bus macros. Instead, dedicated FFs in the static region serve as anchor points between the static and dynamic regions. As part of DPSR-LD, we have implemented an ICAP controller, a Spartan-6 bitstream compressor (software), and the decompressor (embedded in our ICAP controller) in order to reduce PR time. Our final contribution is to eliminate the resource overhead that comes from PUFs (i.e. PUFs are placed in dynamic regions which are then reclaimed for the actual (protected) design). The efficiency and practicality of the methods were demonstrated by proof-of-concept implementation of a GPIO controller design on Spartan-6 (XC6SLX45) using Digilent Atlys Boards.

References


Fig. 9. FPGA Editor screenshots of the dynamic region: Configuration B (Left image) and Configuration A (Right image).


Sezer Gören received her BS and MS degrees in EE from Bosphorus University, Turkey, and a PhD degree in CMPE from University of California, Santa Cruz, USA. She worked as a senior engineer in Silicon Valley from 1998 until 2005 at Cadence Design Systems, Apple Computer, PMC-Sierra, and Aarohi. She is currently an associate professor in the CMPE Department at Yeditepe University in Istanbul. Her research interests include electronic design automation, design verification, test, reconconfigurable computing, and embedded system design.

Ozgur Ozkurt received his BS and MS degrees in EE from Bahcesehir University, Istanbul, Turkey. He has been with Vestek R&D, Istanbul, Turkey.

Abdullah Yildiz received his BS and MS degrees in EE from Uludag University and Ozyegin University, respectively. He is currently working on his PhD degree in CMPE at Yeditepe University.

H. Fatih Ugurdag is an assistant professor in the EE Department at Ozyegin University in Istanbul. He received his MS and PhD in EE from Case Western Reserve University in Cleveland, USA. He received his BS in EE and Physics from Bosphorus University in Turkey. He worked in the industry between 1989-2004 at North Coast Automation, GM, Lucent, Juniper, Pulcent, ARC, and Nvidia. His research interests are diverse and include reconconfigurable computing, computer arithmetic, design automation, and real-time video processing.

Rajat Subhra Chakraborty is an assistant professor in the CSE-CMPE Department of IIT Kharagpur. He has a PhD in CMPE from Case Western Reserve University. His professional experience includes a stint as CAD Software Engineer at National Semiconductor, and a graduate internship at Advanced Micro Devices. His research interest includes hardware security, including design methodology for hardware IP/IC protection, hardware Trojan detection/prevention through design and testing, attacks on hardware implementation of cryptographic algorithms, and reversible watermarking for digital content protection.

Debdeep Mukhopadhyay received his BTech degree in EE from IIT Kharagpur, India. He obtained his MS and PhD in CSE-CMPE Department of IIT Kharagpur. He is presently working as an assistant professor in the CSE-CMPE Department of IIT Kharagpur. His research interests include cryptography, VLSI of cryptographic algorithms and side channel analysis. He is the recipient of Indian Semiconductor Association Techno Inventor, Indian National Science Academy Young Scientist and Engineer awards, Associate of Indian Academy of Science and Indo-US fellowships.