Physical Unclonable Functions and Dynamic Partial Reconfiguration for Security in Resource-Constrained Embedded Systems

Abstract

Authentication and encryption within an embedded system environment using cameras, sensors, thermostats, autonomous vehicles, medical implants, RFID, etc. is becoming increasing important with ubiquitious wireless connectivity. Hardware-based authentication and encryption offer several advantages in these types of resource-constrained applications, including smaller footprints and lower energy consumption. Bitstring and key generation implemented with Physical Unclonable Functions or PUFs can further reduce resource utilization for authentication and encryption operations and reduce overall system cost by eliminating on-chip non-volatile-memory (NVM). In this paper, we propose a dynamic partial reconfiguration (DPR) strategy for implementing both authentication and encryption using a PUF for bitstring and key generation on FPGAs as a means of optimizing the utilization of the limited area resources. We show that the time and energy penalties associated with DPR are small in modern SoC-based architectures, such as the Xilinx Zynq SoC, and therefore, the overall approach is very attractive for emerging resource-constrained IoT applications.

1 Introduction

The proliferation of embedded systems in the expansion of Internet-of-things (IoT) to 10's of billions of connected devices has vastly widened the attack surface for adversaries targeting the theft of information and/or the malicious, sometimes destructive, control of such systems. However, the area and power consumption overheads associated with adding security functions such as authentication and encryption to protect such systems can be large relative to overall system size. Moreover, the approach taken to secure secret information in desktops and other supervised systems is not adequate for embedded systems. Embedded systems are more vulnerable to invasive attacks because many times they are deployed to unsupervised, remote environments. Attackers can mount more sophisticated attacks when physical access is possible, using low-cost, but highly effective, bench-top test and measurement equipment. The basis of security in a typical embedded system is a stored 'secret', e.g., a master key that can be used to generate session keys and other bitstring identifiers for security functions. Most attacks focus on learning this stored secret, either through side-channel attacks or other methods designed to invasively probe the chip.

The most common approach to storing 'secrets' used for authentication and encryption functions on the chip is through the use of non-volatile memory (NVM). Although NVM is highly reliable, it adds costs to products because of the additional masks required during chip fabrication, and it is vulnerable to invasive attacks [1]. Physical Unclonable Functions (PUFs) have been proposed as an alternative to NVM key storage, and for generating unique and untrackable authentication information. PUFs leverage small, immutable variations that occur in the manufacturing process of otherwise identical digital chips as a means of deriving unique and repeatable bitstrings that can serve as encryption keys or as chip-specific identifiers in the message exchanges during authentication operations. The automatic, on-chip generation of keys and bitstrings also simplifies and strengthens key management by removing human intervention in the key generation process, and eliminates other production-floor processes required to transfer and write them into NVMs. PUFs require no digital storage mechanism, i.e., bitstrings and keys are generated on-the-fly as needed, and are also *tamper-evident*, whereby attempts by adversaries to invasively read-out PUF data can often irreversibly change and/or destroy that data.

In this paper, we propose an FPGA-based resource-constrained architecture that reduces area overhead by reprogramming a region in the programmable logic using *dynamic partial reconfiguration* (DPR) first with a PUF-based bitstring/key generation and authentication module called HELP [2] and then with a AES encryption module [3]. DPR is an advanced FPGA feature that allows a portion of the reprogrammable fabric of the FPGA to be reconfigured on-the-fly while the remainder of the FPGA remains unchanged and fully operational. One of the benefits of DPR is to reduce resource requirements within the FPGA, which reduces system cost and size. DPR can also potentially reduce energy consumption because of the smaller leakage power associated with the smaller FPGA. However, the reduced leakage is partially offset by the power required to dynamically reconfigure the device. In the proposed architecture, the DPR operation is only performed once per authentication-encryption session, and therefore the energy overhead is amortized over the number of messages that are encrypted.

The ARM Cortex A-9 microprocessor running a version of embedded Linux on a Xilinx Zynq 7020 FPGA is used as the test bed for the DPR experiments. A C program and VHDL implementation of HELP are used to demonstrate a fully operational PUF-based, privacy-preserving, mutual authentication protocol and data encryption application.

The experimental evaluation carried out in this work uses HELP and a PUF-based authentication protocol [2][4]. HELP provides several benefits within the proposed archi-

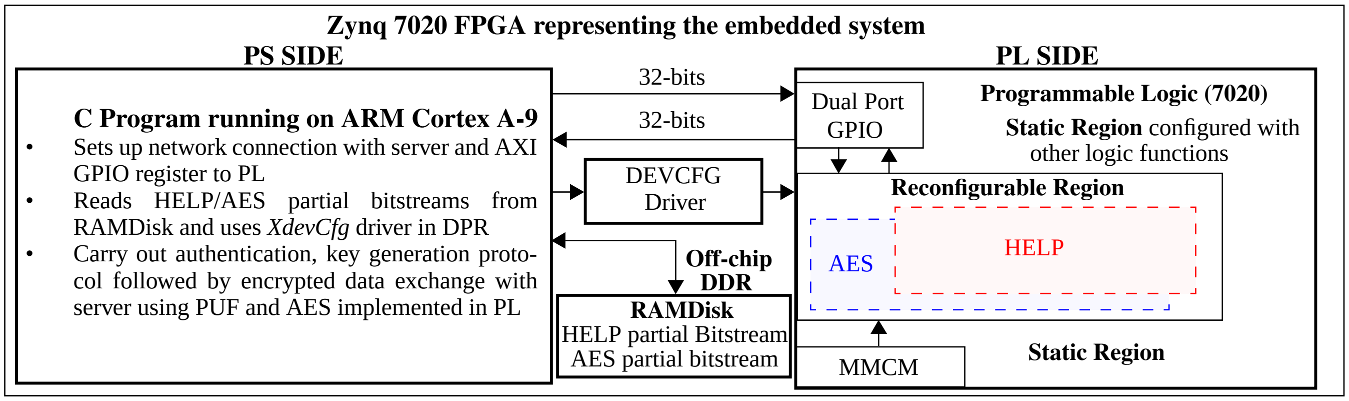

Zynq 7020 FPGA representing the embedded system
PS SIDE
PL SIDE
C Program running on ARM Cortex A-9
• Sets up network connection with server and AXI GPIO register to PL
• Reads HELP/AES partial bitstreams from RAMDisk and uses XdevCfg driver in DPR
• Carry out authentication, key generation protocol followed by encrypted data exchange with server using PUF and AES implemented in PL
32-bits
32-bits
DEVCFG Driver
Off-chip DDR
RAMDisk
HELP partial Bitstream
AES partial bitstream
Dual Port GPIO
Programmable Logic (7020)
Static Region configured with other logic functions
Reconfigurable Region
AES
HELP
MMCM
Static Region

Fig. 1. Overall system architecture.

tecture. First, the version investigated here uses an implementation of the AES SBOX as the functional unit from which it derives random information for bitstring generation. Second, HELP is able to produce an exponential number of bitstring responses, which is critically important to the security properties of PUFs which use unprotected interfaces for authentication. This small footprint version of HELP is demonstrated in a Xilinx Zynq 7020 FPGA SoC but can also be used in smaller resource-constrained systems.

This paper is organized as follows. Related work is described in Section 2 and Section 3 presents the system architecture and synthesis flow. Section 4 reports on the experiments carried out as well as system characteristics and runtimes. Section 5 gives Conclusions.

2 Related Work

Papers [5-6] describe the application of dynamic partial reconfiguration for switching between different versions of the AES engine. They dynamically switch the AES core between 128-, 192- and 256-bit implementations assuming the application will need different versions. The authors of [6] investigate a system that incorporates AES-256 and SHA-3 as reconfigurable modules for applications related to wireless sensor nodes but they do not implement authentication, nor do they incorporate a PUF for key generation and therefore are more costly and vulnerable to attacks. The authors of [7] propose the use of partial reconfiguration in wireless sensor nodes for authentication and encryption, but do not incorporate a PUF-based key generator.

3 System Architecture

A block diagram of the overall system architecture is shown in Fig. 1. The Xilinx Zynq 7020 is an SoC with both a ‘processor system’ (PS) component shown on the left and a ‘programmable logic’ (PL) component shown on the right. The PS component includes two ARM Cortex A-9 processors plus an AXI interconnection network, instruction and data caches and other processor support modules. The PL component includes a programmable fabric and a variety of embedded IP blocks such as Block RAMs (BRAMs), multipliers, etc.

We built a Linux kernel using source code from the *git* repository [8] for the Zedboard [9] with default options including TCP/IP network support and General purpose I/O support (GPIO). Xilinx Vivado [10] is used to create a mixed IP/VHDL project (discussed below) that is then exported to SDK as the base hardware platform for software development. SDK is used to compile a custom C program and to configure a *boot* file for the embedded Linux OS. An SD card is created with the Linux kernel *zImage* and boot file *BOOT.bin* plus support files *RAMDisk* and *devicetree.dtb* for use with the Zedboard. The embedded Linux operating system (OS) provides secure-socket-shell (ssh) and session control protocol (scp) network programs for remote sessions and file transfer, resp. between the Zedboard and a host computer.

The custom C program referenced above carries out the following operations:

- Sets up an unencrypted *socket* connection with a host computer for communications and data transfer. Sets up a memory-mapped register interface through Xilinx AXI GPIO for data transfer between the PS and PL sides.
- Transfers HELP and AES partial bitstream files used by DPR from the host and stores them into a portion of the 256 MB DDR on the Zedboard configured as a RAMDisk under Linux (note: these files can also be stored and accessed from the SD card or other bitstream dedicated memory available to the system).
- Reads partial bitstreams from DDR and carries out DPR using the Xilinx *XDevCfg* interface to a dedicated reconfiguration region in the PL side.
- Controls the key generation, authentication and then encryption protocols running in PL as a demonstration of a typical IoT session carried out between an embedded system and a secure server.

As is customary in a fielded FPGA-based embedded system, we assume that Xilinx security mechanisms are used to encrypt the static bitstream and DPR bitstreams for HELP and AES to prevent reverse engineering and tamper.

3.1 HELP PUF and Authentication Protocol

Fig. 2 shows a block level diagram of the HELP engine and the portion of the authentication protocol that runs on the *hardware token* in the shaded region on the left. An abstraction of the AES S-BOX functional unit is shown as a gate-level netlist on the far left. The VHDL modules that implement the HELP engine include a *Challenge Selection module*, which is responsible for selecting and applying 2-vector sequences (challenges) delivered to the hardware token through a network connection or read out from an on-board or on-chip memory. The *Clock Strobe module* controls the fine phase shift feature in the Xilinx on-chip digital clock manager (MMCM) to enable accurate measurements of path delays. The path delays are labeled as *Output Responses* in

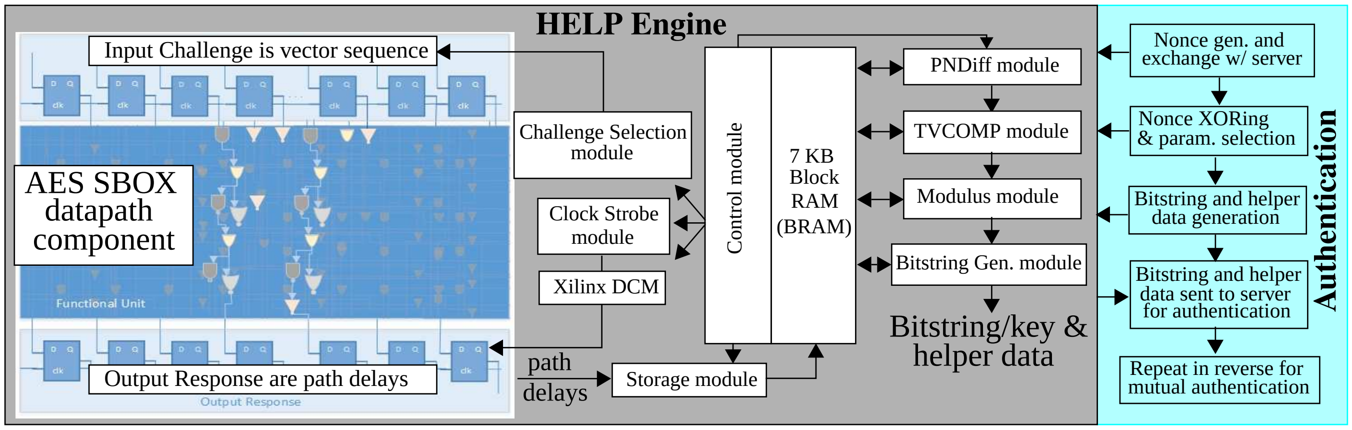

HELP Engine
Input Challenge is vector sequence
AES SBOX datapath component
Functional Unit
Output Response are path delays
Output Response
Challenge Selection module
Clock Strobe module
Xilinx DCM
path delays
Storage module
Control module
7 KB Block RAM (BRAM)
PNDiff module
TVCOMP module
Modulus module
Bitstring Gen. module
Bitstring/key & helper data
Nonce gen. and exchange w/ server
Nonce XORing & param. selection
Bitstring and helper data generation
Bitstring and helper data sent to server for authentication
Repeat in reverse for mutual authentication
Authentication

Fig. 2. HELP Engine (left) and authentication protocol (right).

the figure, and are obtained by timing transitions propagating through the SBOX to the FFs connected to its outputs. A *Storage module* saves a digitized representation of the path delays in an on-chip Block RAM (BRAM). Once a set of 4096 path delays are obtained, the *Control module* processes the timing values into a **bitstring** and/or **key** and **helper data** using a sequence of parameterized modules named *PNDiff*, *TVCOMP*, *Modulus* and *Bitstring Gen*. The bitstring and key are used for authentication and encryption operations, resp. The helper data is transmitted and stored by a server, and is used by the hardware token to reproduce the bitstring/key when needed by the application while operating in the field (**regeneration**). The helper data is public information and does not need to be encrypted. Details regarding the specific operations carried out during bitstring/ key generation can be found in [4].

The authentication protocol shown on the far right of Fig. 2 is layered on top of the HELP engine. For example, the first module of authentication collects a *nonce* (a random number) from the HELP engine. This nonce is XORed with a server-generated nonce and is used to specify a set of parameters for the HELP engine modules. The exchange and XORing of token-server nonces prevents adversaries from controlling the HELP engine parameters directly in model-building attacks.

The HELP engine is then run to generate a bitstring and helper data as described above. The bitstring and helper data are transmitted to the server. The server authenticates the token by comparing the received bitstring with a bitstring that it computes using the same nonce-selected parameters and path delays it stores from an earlier enrollment operation. Enrollment refers to a special process performed in a secure environment (before the hardware token is deployed into the field) and involves running the HELP engine to obtain the digitized path delay values for storage in the server's database.

The HELP engine also has an encryption mode designed to generate a secret key. The protocol in this case transmits only the helper data to the server. The server also generates helper data using its stored enrollment data that is transmitted to the token. The token and server bitwise AND the helper data bitstrings and use them to generate the secret keys. The AND'ed helper data bitstrings increase the probability that both the server and token generate the same key (see [4] for details). The encryption key is stored for later use by the AES-128 module, which is programmed into the same region as the HELP engine using dynamic partial reconfiguration (DPR). The experimental results presented in Section 4 use these authentication and key generation protocols to characterize performance.

3.2 DPR Synthesis Flow

The Xilinx Zynq 7020 SoC used in our experiments is ideally suited for applications that requires a hardware/software co-design approach. A C program running on the PS side can be designed to implement higher layer components of the embedded system application, such as components for managing network connections, while VHDL can be used on the PL side to implement the application's speed- and security-sensitive operations. We followed this partitioning strategy in the design of the experimental system.

We designed HELP and a 128-bit pipelined version of AES [3] as reconfigurable modules (RMs) in VHDL using the *partial reconfiguration* (PR) design flow defined within Xilinx Vivado's non-project mode of operation [11]. The details of PR process are shown by a flowchart in Fig. 3 (derived from [11]). The overall flow is bottom-up, i.e., the static module as well as the RMs used in the DPR operations are individually synthesized and respective *checkpoints* are saved to enable the complete system module to be built in the last phase. The RM configurations are synthesized in *out_of_context* mode to prevent the synthesis tool from mapping their I/Os to chip I/O pads. The global clock buffers are excluded from the RMs, and are instead instantiated in the static (top) module, because they cannot be a component of partial reconfiguration.

A *blackbox* (PR region) is added to the static module as a placeholder for the RMs. The RMs are then individually synthesized and loaded into the *blackbox*. The *HD.Reconfigurable* is set on the *blackbox* to inform the synthesis tool that this region is designated for DPR. Manual floor planning of the PR region can now be carried out, which involves designating a region in the PL side and adjusting its position and size to accommodate the logic in the RM modules. Additional properties including *snapping mode* and *reset_after_reconfiguration* can also be enabled on the PR region. *Snapping mode* ensures that boundaries to the PR region are legal, while *reset_after_reconfiguration* ensures

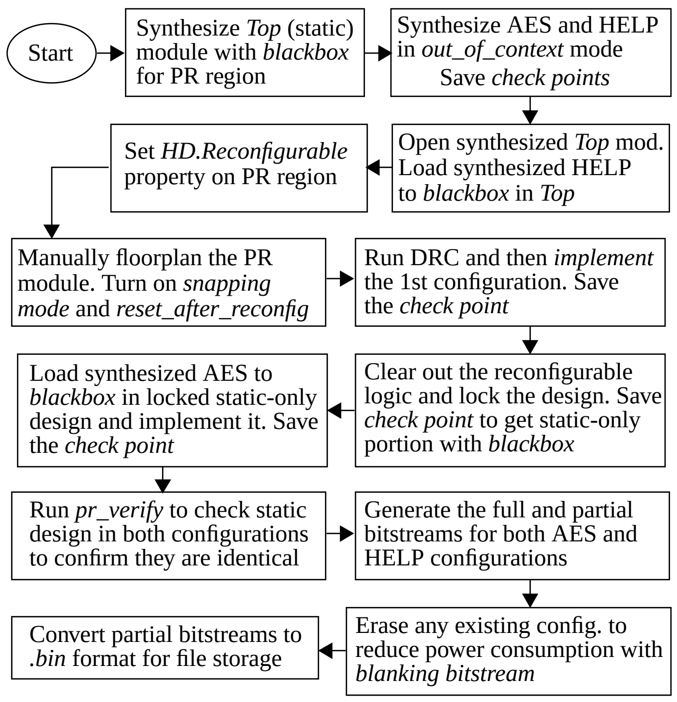

Start
Synthesize *Top* (static) module with *blackbox* for PR region
Synthesize AES and HELP in *out_of_context* mode Save *check points*
Open synthesized *Top* mod. Load synthesized HELP to *blackbox* in *Top*
Set *HD.Reconfigurable* property on PR region
Manually floorplan the PR module. Turn on *snapping mode* and *reset_after_reconfig*
Run DRC and then *implement* the 1st configuration. Save the *check point*
Clear out the reconfigurable logic and lock the design. Save *check point* to get static-only portion with *blackbox*
Load synthesized AES to *blackbox* in locked static-only design and implement it. Save the *check point*
Run *pr_verify* to check static design in both configurations to confirm they are identical
Generate the full and partial bitstreams for both AES and HELP configurations
Erase any existing config. to reduce power consumption with *blanking bitstream*
Convert partial bitstreams to *.bin* format for file storage

Fig. 3. DPR synthesis flow.

the FFs in the PR region are initialized after each DPR operation.

After performing a *design-rule-check* (DRC), the implementation step of the Vivado tool flow is run. The process is repeated for the second RM. A process step called *pr_verify* is then performed which ensures that the static portion of the design is same in both instantiations. If *pr_verify* is successful, partial and full programming bitstreams can then be generated and converted into binary files. A optional *blanking bitstream* can be configured into the PR region as a means of reducing power consumption when the PR region is not being used. The *blanking bitstream* deletes the configuration of the PR region and adds buffer ports to its I/Os.

The physical layout of experimental system is shown in Fig. 4 as a screen snapshot of the view provided by Xilinx Vivado's Implementation View (VIV). The PR region is defined manually using VIV as a purple rectangle and is the region in which the AES and HELP RMs are instantiated using the DPR process. The PR region occupies approx. 5% of the PL region. The static portion of the design is instantiated in a small part of the remaining 95% and includes the digital clock manager (MMCM) and AXI GPIO register interfaces to the PS side. The PS side with ARM Cortex A-9 microprocessors, is depicted in the upper left of the image.

3.3 Runtime Operation

The static portion of the PL is programmed upon boot-up of the hardware token. The C program running is started manually in our experiments but can be configured to run automatically once Linux boots, or run immediately in 'bare metal' applications that do not include an operating system. Our version of the C program performs three basic tasks but would normally also encapsulate other application-related functions required of the embedded system. The C program in our experiments 1) reads and loads the RMs (that have

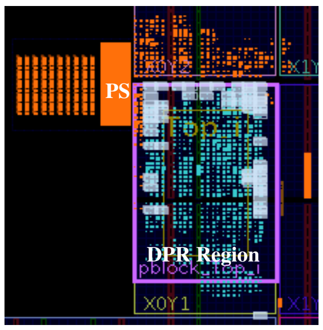


Fig. 4. FPGA editor view of system implementation.

been stored as files in the RAMDisk component of the off-chip DDR), 2) sets up a memory mapped interface to the AXI GPIO registers and 3) configures a network socket connection with a host machine (server). The C program uses system calls to the Linux driver module *Xdevcfg* to load the RMs. *Xdevcfg* accesses the DPR port on the PL side (PCAP interface) to dynamically load the partial bitstream of the RM. Data transfer between the C program and the RMs (at the PS-PL interface) is accomplished using AXI general-purpose I/O (GPIO) registers, configured with two 32-bit channels for input and output, resp.

As described in Section 2, a privacy-preserving, mutual authentication operation is carried out with the server, followed by an AES encrypted session as a means of demonstrating a working system and for collecting runtime statistics about its performance. Fig. 5 gives a flowchart of the entire sequence of events, from boot-up through DPR of HELP, token authentication, mutual authentication, key generation, encryption, transmission and decryption on the server. Once the session ends, a final DPR step is performed (not shown) that configures the PR region with the *blanking bitstream* and the system enters low power mode.

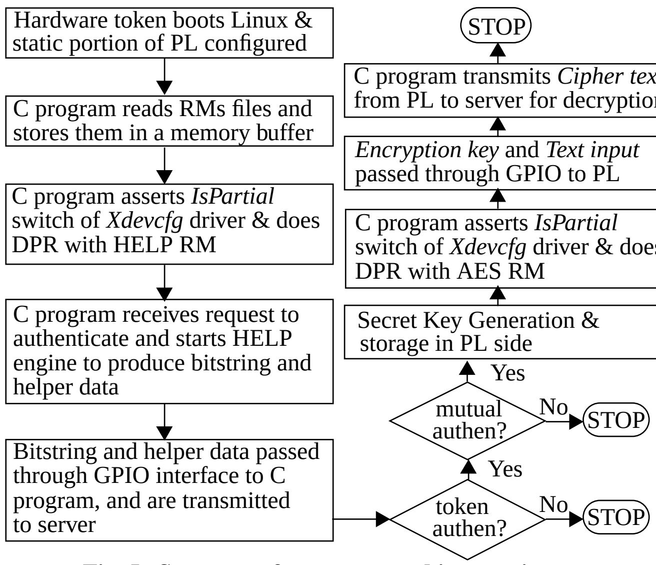
4 Experimental Results

4.1 Area and Runtime Overheads of DPR Version

Table 1 shows the resources used by the HELP engine and the OpenCores pipelined version of AES-128 [3] when the RM synthesis described in Section 3.2 is carried out. Note that the static portion also includes an MMCM which cannot be included in the DPR region. The DSP component used by the HELP engine is a 25-bit multiplier.

The number of LUTs determines the smallest sized PR region that can accommodate the RMs. The AES component is slightly larger and therefore sets the minimum size of the reconfigurable region. The best case area saving is achieved when the number of LUTs for both RMs is the same, yielding an area reduction of 50% for the logic placed in the PR region. The similarity in size of the actual RMs produces a value close to the optimal at 2,321/(2,321+2,680) = 46.4%.

Table 2 shows the runtimes associated with the DPR operation only, which accounts for the time spent reading the RM bitstreams as files from the RAMDisk and processing one of them using system calls to the *Xdevcfg* driver. DPR runtime is proportional to the size of the RM bitstream, which are approx 313 KB.



Hardware token boots Linux & static portion of PL configured
C program reads RMs files and stores them in a memory buffer
C program asserts *IsPartial* switch of *Xdevcfg* driver & does DPR with HELP RM
C program receives request to authenticate and starts HELP engine to produce bitstring and helper data
Bitstring and helper data passed through GPIO interface to C program, and are transmitted to server
token authen?
No
STOP
Yes
mutual authen?
No
STOP
Yes
Secret Key Generation & storage in PL side
C program asserts *IsPartial* switch of *Xdevcfg* driver & does DPR with AES RM
Encryption key and *Text input* passed through GPIO to PL
C program transmits *Cipher text* from PL to server for decryption
STOP

Fig. 5. Sequence of steps executed in experiments.

Table 1: Resource Utilization of DPR version.

Component	**LUTs** of 53,200	**FFs** of 106,400	**MUX** of 26600	**BRAM** of 512KB	**DSP** of 220
HELP	2321 (4.3%)	979 (0.9%)	53 (0.2%)	7 KB (1%)	1 (0.5)
AES	2680 (5.03%)	629 (0.6%)	256 (0.96%)	0 (0%)	0 (0%)
Static Portion	625 (1.2%)	972 (0.9%)	0 (0%)	0 (0%)	0 (0%)

Table 2: DPR runtime.

Activity	**Time (µs)**
Loading partial bitstreams from file system	663
Open "IsPartialBitstream" attribute of *Xdevcfg*	74
Open the *Xdevcfg* driver	19
Writing partial bitstream to PL from PS using *Xdevcfg*	4535
Totals	5291

Table 3: Protocol runtimes of DPR version.

Activity	**Time (us)**
Partial reconfiguration time: HELP	5291
Time to receive challenge vectors from the server	7115
Token authentication time	810768
Server authentication (mutual) time	799852
Dynamic Key Generation time	783506
Partial reconfiguration time: AES	5285
First 128-bit encryption time	10229
Total Time	2.4 seconds

Table 3 gives the runtimes for components of the authentication, key generation and encryption protocols, which includes two instances of the DPR runtime from Table 2. The time overhead associated with DPR is approx. 11 ms or 0.5% of the total runtime. Note that the time overhead penalty decreases as the number of encryptions increases, i.e., the 0.5% applies only to the case when only two 128-bit encryptions are performed as shown.

The time overhead associated with the authentication and key generation operations is given by the sum of rows two, three, four and five in Table 3 as approx. 2.3 seconds. This runtime overhead is a one-time penalty associated with each authentication-encryption session, and includes all network delays associated with nonce and bitstring exchanges between the hardware token and server. Both the DPR and bitstring/key generation overheads are small enough to suit many common types of IoT applications including those developed for home automation and sensor networks.

4.2 Energy Overhead of DPR Version

The energy consumption of the protocol is determined by measuring the voltage drop across a 9 Ohm resistor placed in series with V_{DD} on the Zedboard's input power cable. The input voltage from the wall-plug mounted regulator is 12 volts. A MAXIM on-board voltage regulator chip down-converts the input 12 volts to a set of lower voltages, one of which drives the 1.00 volt core power supply pins of the Zynq 7020 chip. The low energy requirements of the implementation made it impossible to use the Zedboard power monitoring pins installed on the 1.00 volt side of the MAXIM chip.

The voltage transient difference waveform shown in Fig. 6 is collected using a differential measurement with two active oscilloscope probes placed on either side of the inserted resistor. The C program is configured to drive a trigger output signal connected to one of the PMOD output pins on the Zedboard, which is used as the input trigger to the oscilloscope. The trigger signal is de-asserted immediately after each of the major steps in the protocols are completed. A *usleep* system call with a duration of 50 ms is inserted after the de-assertion trigger event to make it easy to identify the beginning and end of the voltage transient associated with the protocol step. The trigger waveform is shown along the bottom of Fig. 6, and the corresponding voltage transient difference waveform shown above it is labeled with each of the major steps of the protocol. Measurement noise is reduced by computing an average from a sequence of 100 runs of the protocol.

Table 4: Energy consumption of DPR version

Activity	**Energy (mJ)**
Partial reconfiguration energy: HELP	1.85
Energy to receive challenge vectors from the server	0.20
Token authentication energy	60.10
Server authentication (mutual) energy	44.10
Encryption Key Generation	61.60
Partial reconfiguration energy: AES	1.90
Encryption energy (256 bits using 128-bit version)	0.07
Total Energy	169.82

The current drawn by the Zedboard when the protocol is not running is computed from the baseline voltage drop using Ohms law as 4.32/9.0 Ohms = 480 mA. The energy used in each protocol step is derived from the area under the voltage difference waveform. The area for the first DPR step is calculated as 0.001388 V-s. Dividing through by the resis-

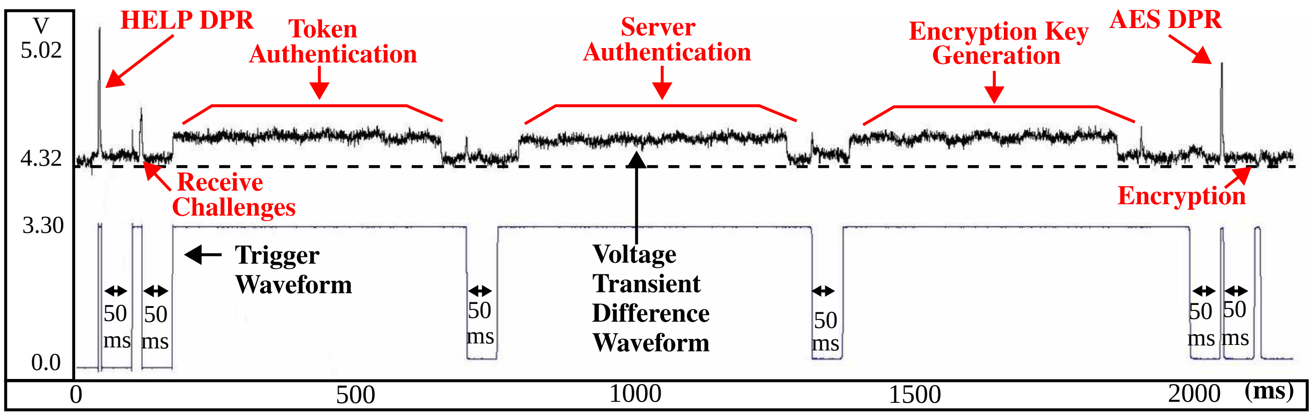


Fig. 6. Power Consumption Waveform

tance yields 0.0001542 A-s and multiplying by 12 volts gives the energy consumption, i.e.,1.85 mJ. Table 4 lists the energy consumption for each of the major protocol steps using the same method carried out over the other regions of the voltage transient difference waveform. The DPR operations are only performed once for each session and therefore, assuming each session carries out 100's of encryption operations, the overhead of DPR will be much smaller in the overall energy consumption profile in an actual usage scenario.

4.3 Wireless Interface

Wireless interfaces are commonly used in IoT applications. We tested the proposed authentication/encryption system using a Belkin N-300 USB WiFi adaptor [12] connected to the Zedboard using USB-OTG (On-the-Go) connector. A custom Linux kernel (*zImage*) is cross-compiled for the ARM μprocessor as a means of incorporating the Realtek RTL8712u driver which supports the adaptor. A loadable module *r8712u.ko* is also compiled and configured with firmware *r8712u.bin* [13]. A bootable system is created by copying the module and firmware to /lib/modules and /lib/firmware/rtlwifi respectively to the Linux root file system stored in *RamDisk*. We also needed to download the Linux wireless configuration utility *iwconfig* from the Hewlett Packard (PH) github [14]. Performance tests run using the wireless interface produced nearly identical results to those presented using the wired interface described in the previous sections. Therefore, the speed, power and area results presented are relevant for wireless IoT applications.

5 Conclusions

An FPGA-based experimental evaluation using dynamic partial reconfiguration (DPR) is carried out using a PUF based authentication and AES encryption algorithm, both implemented in VHDL and designed as reconfigurable modules (RMs). A hardware-embedded delay PUF called HELP is used to generate authentication bitstrings and encryption keys using the AES SBOX data path component as its source of random information. An area reduction of nearly 50% is possible using a DPR strategy on these hardware modules with only a small runtime and energy penalties, illustrating the practical value of using DPR for security related functions in resource-constrained embedded systems.

6 References

[1] S. Kannan, N. Karimi, O. Sinanoglu and R. Karri, "Security Vulnerabilities of Emerging Nonvolatile Main Memories and Countermeasures," *IEEE Transcations on Computer Aided Design of Integrated circuits and Systems*, Vol. 34, No. 1, January, 2015.
[2] J. Aarestad, P. Ortiz, D. Acharyya and J. Plusquellic, "HELP: A Hardware-Embedded Delay-Based PUF", *Design and Test of Computers*, Mar., 2013, pp. 17-25.
[3] http://opencores.org/project,aes_pipe
[4] W. Che, F. Saqib and J. Plusquellic, "A Privacy-Preserving, Mutual PUF-Based Authentication Protocol", accepted with minor revisions, http://www.mdpi.com/journal/cryptography/special_issues/physical_security, 2016.
[5] Z. E. Abidine A. Ismaili and A. Moussa, "Self-Partial and Dynamic Reconfiguration Implementation for AES using FPGA", *IJCSI International Journal of Computer Science Issues*, Vol. 2, 2009.
[6] S. Wankhade and R. Mahajan, "Performance Enhancement of AES Algorithm Using Dynamic Partial Reconfiguration", *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, Vol. 3, Issue 4, April 2014.
[7] L.A. Cardona, B. Lorente and C. Ferrer, "Partial Crypto-Reconfiguration of nodes based on FPGA for WSN", *International Carnahan Conference on Security Technology* (ICCST), 2014.
[8] https://github.com/Digilent/linux-digilent.git
[9] https://zedboard.org/
[10] http://www.xilinx.com/products/design-tools/vivado.html
[11] www.xilinx.com/support/documentation/sw_manuals/xilinx14_1 ug702.pdf
[12] www.belkin.com/us/support-product?pid=01t80000002vzbVAAQ
[13] https://git.kernel.org/cgit/linux/kernel/git/firmware/linux-firmware.git
[14] https://hewlettpackard.github.io/wireless-tools/Tools.html