# Self-Authenticating Secure Boot for FPGAs

G. Pocklassery+, W. Che+, F. Saqib*, M. Areno^ and J. Plusquellic
Enthentica+, University of North Carolina*, Trusted and Secure Systems^, University of New Mexico

## Abstract

*Secure boot within an FPGA environment is traditionally implemented using hardwired embedded cryptographic primitives and NVM-based keys, whereby an encrypted bitstream is decrypted as it is loaded from an external storage medium, e.g., Flash memory. A novel technique is proposed in this paper that self-authenticates an unencrypted FPGA configuration bitstream loaded into the FPGA during start-up. The power-on process of an FPGA loads an unencrypted bitstream into the programmable logic portion which embeds the self-authenticating PUF architecture. Challenges are applied to the components of the PUF engine both as a means of generating a key and performing self-authentication. Any modifications made to the PUF architecture results in key generation failure, and failure of subsequent stages of the secure boot process. The generated key is used in the second stage of the boot process to decrypt the programmable logic portion of the design as well as components of the software, e.g., Linux operating system and applications, that run on the processor side of the FPGA.*

## 1 Introduction

SRAM-based FPGAs need to protect the programming bitstream against reverse engineering and bitstream manipulation (tamper) attacks. Fielded systems are often the targets of attack by adversaries seeking to steal intellectual property through reverse engineering, or attempting to disrupt operational systems through the insertion of kill switches known as hardware Trojans. Internet-of-things (IoT) systems are particularly vulnerable given the resource-constrained and unsupervised nature of the environments in which they operate.

FPGAs implementing secure boot usually store an encrypted version of the programming bitstream in an off-chip non-volatile memory (NVM) as a countermeasure to these types of attacks. Modern FPGAs provide on-chip battery-backed RAM and/or fuses for storage of a decryption key, which is used by vendor-embedded encryption components within the FPGA to decrypt the bitstream as it is read from the external NVM during the boot process [1]. Recent attack mechanisms that are able to read out on-chip stored keys therefore threaten the security of the boot process [2].

In this paper, we propose a PUF-based key generation strategy that addresses the vulnerability of on-chip key storage. Moreover, the proposed secure boot technique is self-contained in that none of the vendor-embedded security primitives are utilized. We refer to the system as Self-Authenticated Secure Boot (SASB). SASB uses a PUF implemented in the programmable logic (PL) side of an FPGA to generate the decryption key at boot time, and then uses it for decrypting an off-chip bitstream, i.e., the second stage boot image, which can contain PL components as well as software components such as an operating system and applications. The decrypted PL components are programmed directly into the unused portion of the PL side using dynamic partial reconfiguration while the software components are loaded into DRAM for access by the processor system (PS). The decryption key is destroyed once this process completes, minimizing the time the decryption key is available.

The following are key contributions of the proposed system. First, during enrollment at a secure facility, the PUF within SASB is configured to measure the path delays through components of the SASB implementation as a means of generating an encryption key, that is then used to encrypt the second stage boot image, i.e., the encrypted bitstream. Second, during regeneration in the field, the PUF regenerates the same key while simultaneously self-authenticating the SASB bitstream. This architecture is self-authenticating because any tamper with the existing SASB implementation will change the delay characteristics of one or more paths, which in turn, will introduce bit-flip errors in the regenerated key. Failure to regenerate the enrollment key prevents the system from booting.

The most significant threat to the proposed system is an attack in which an adversary adds additional functions to the unused portion of the PL fabric in the unencrypted SASB bitstream. Fanout can be easily added to the routing networks defined by the FPGA switch boxes, providing multiple opportunities for adversaries to add 'observation points' to, e.g., the AES key registers as a means of creating an information leakage channel. We discuss strategies in the design of SASB that create custom paths through the FPGA routing switch boxes. These custom paths, called ***blocking paths***, are designed to block all fanout points to wires which carry 'sensitive' information, e.g., wires driven by the key register. The delays of each of the *blocking paths* is also measured and used in the key generation process. Therefore, adversaries who remove or manipulate the configuration of the *blocking paths* causes key regeneration to fail.

The SASB key generation algorithm is designed such that adversarial modifications that cause path delays to change beyond a threshold cause an avalanche effect, i.e., one path delay that exceeds the threshold causes a large fraction of the key bits to change. This feature is designed to prevent adversaries from carrying out multiple, incremental attacks which target one key bit (or small subsets) at a time.

The rest of this paper is organized as follows. A survey

978-1-5386-4731-8/18/$31.00 

of related work is provided in Section 2. An overview of the existing Xilinx boot process is provided in Section 3. Section 4 provides an overview of the proposed SASB system and discusses various attack models on the proposed system. Section 5 describes important concepts of the SASB system. Conclusions are provided in Section 6.

## 2 Background

Bitstream reverse engineering represents a vulnerability to FPGAs. While FPGA companies introduce advanced methods to encrypt and authenticate bitstreams, reverse engineering and fault injection attacks continue to evolve. Reference [3] discusses methods to detect and manipulate cryptographic components such as S-boxes in the bitstream, by reverse engineering LUT programming, thereby weakening cryptographic primitives. Automatic extraction of secret keys from the FPGA using bitstream fault injection attacks is the focus of [4]. Reference [5] proposes a method to weaken hardware cryptography engines by the insertion of Trojans into low-cost FPGAs (Spartan3-E). The authors of [6] and [7] discuss several ways to extract cryptographic keys stored in NVMs.

Battery Backed RAMs (BBRAM) and E-Fuses are also used for storing keys in FPGAs. While BBRAMs do not suffer from the same security issues as those related to NVM, the requirement for a battery and its limited lifetime complicate system design. E-Fuses are one-time-programmable (OTP) and therefore reduce flexibility in key management, and depending on their design, can also be inspected using semi-invasive attacks such as device de-processing and inspection via scanning electron microscopes (SEM) [7]. The HELP PUF is used in this work to address the vulnerabilities associated with conventional key storage mechanisms [8].

## 3 Overview of Secure Boot under Xilinx

Xilinx FPGA vendors incorporate an on-chip 256-bit AES decryption engine to protect the confidentiality of externally stored bitstreams [1]. The Xilinx tool flow optionally encrypts the bitstream using a randomly generated or user-specified key. The decryption key is loaded via JTAG at a secure facility into a dedicated eFuse NVM or battery-backed BRAM (BBRAM). The in-field boot process determines if the external bitstream includes an encrypted-bitstream indicator and, if so, decrypts the bitstream using cipher block chaining (CBC) mode of AES. During encryption, CBC mode XORs the previous block ciphertext with the next block plaintext before encrypting the current block (decryption reverses this process) as a mechanism to force different ciphertexts for replicated components in the plaintext.

To prevent bit-flip fault injection attacks, Xilinx incorporates data authentication into the boot process [4]. SHA-256 is used to compute a 256-bit keyed hashed message authentication code (HMAC) of the bitstream. The key component allows both the sender of the bitstream to be authenticated and any tamper with the bitstream to be detected with high probability.

The secure boot process proposed by Xilinx is to first compute the HMAC of the unencrypted bitstream, which is then embedded in the bitstream and encrypted by AES. During in-field boot, a second HMAC is computed as the bitstream is decrypted and compared with the HMAC embedded in the decrypted bitstream. If the comparison fails, the FPGA does not become active. The secure boot process provides confidentiality and data integrity and is able to detect transmission failures, attempts to program the FPGA with a non-authentic bitstream and tamper attacks on the authentic bitstream. Public key cryptography is used in modern Xilinx SoC architectures to provide authentication during the secure boot process. The public key is stored in an NVM and used to authenticate configuration files including the First Stage Boot Loader (FSBL). Therefore, public key cryptography provides secondary authentication and primary attestation.

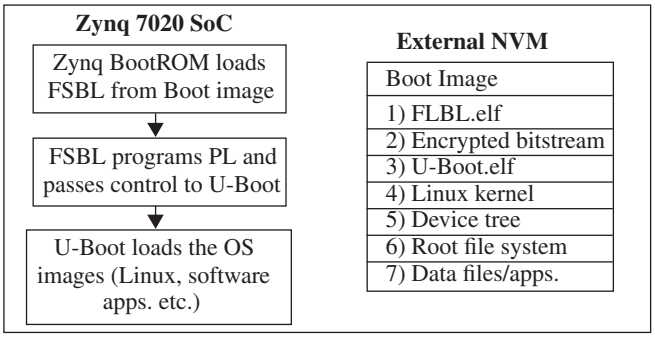


**Fig. 1. Xilinx Zynq SoC boot process.**

The Xilinx Zynq 7020 SoC used in this paper incorporates both a processor (PS) side and programmable logic (PL) side. The processor side runs an operating system (OS), e.g., Linux, and applications on a dual core ARM cortex A-9 processors, which are tightly coupled with PL side through AXI interconnect.

The primary steps of a secure boot process for the Xilinx Zynq SoC is shown in Fig. 1. The Xilinx BootROM loads the FSBL from an external NVM to DDR (DRAM). The FSBL programs the PL side and then reads the second stage boot loader (U-Boot), which is copied to DDR, and passes control to U-Boot. U-Boot loads the OS images, which include a bare-metal application, or the Linux OS, embedded software applications and data files. A secure boot process requires the boot process to begin with a root of trust, and then carry out authentication in each of the subsequent stages of the boot process. As indicated above, Xilinx uses public key cryptography, i.e., RSA, for authentication and attestation of FSBL and other configuration files, and a hardwired 256-bit AES engine and HMAC to securely decrypt and authenticate boot images on chip using a BBRAM or E-Fuse embedded key. The root of trust in the Xilinx proposed secure boot scheme is the embedded key(s), i.e., the security of the entire process depends on keeping the digitally stored, embedded key(s) confidential.

## 4 Overview of Self-Authenticating Secure Boot (SASB)

The secure boot process proposed in this paper does not make use of any of the security features provided within the Xilinx secure boot architecture, i.e., it is a self-contained and is self-authenticating. A flow diagram of the SASB boot process is shown in Fig. 2. The first step is identical to the existing boot process. The PL component that is programmed into the PL side by the FSBL is the unencrypted SASB bit-

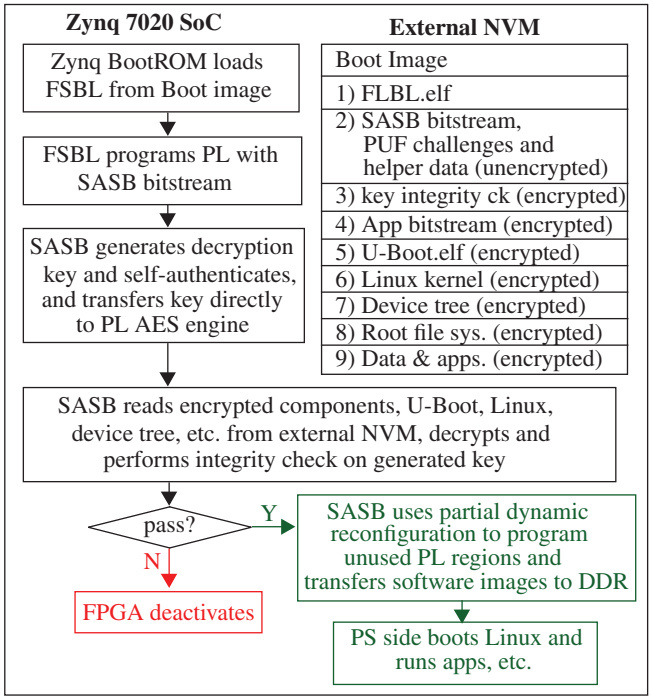


**Fig. 2. Proposed Zynq SoC boot process.**

stream. The FSBL then passes control to SASB and blocks. SASB reads the PUF's challenges and helper data from the external NVM and carries out key regeneration. The key is transferred to an embedded PL-side AES engine. SASB reads the encrypted second stage boot image components labeled as components 3 through 9 in Fig. 2 from external NVM and transfers them to the AES engine.

An integrity check is performed at the beginning of the decryption process as a mechanism to determine if the proper key was regenerated. The first component decrypted is the key integrity check component (labeled 3 in Fig. 2). This component can be an arbitrary string or a secure hash of, e.g., U-Boot.elf, that is encrypted during enrollment and stored in the external NVM. An unencrypted version of the key integrity check component is also stored as a constant in the SASB bitstream. The integrity of the decryption key is checked by comparing the decrypted version with the SASB version. If they match, then the integrity check passes and the boot process continues. Otherwise, the FPGA is deactivated and secure boot fails.

If the integrity check passes, SASB then decrypts components 4 through 9, starting with the application (App) bitstream. An App bitstream (or blanking bitstream if the PL side is not used by the application) is programmed into the unused components of the PL side by SASB using dynamic partial reconfiguration. This ensures that any malicious functions that may have been incorporated by an adversary in unused PL regions of the SASB bitstream are overwritten (more on this later). SASB then decrypts the software components, e.g., Linux, etc. and transfers them directly to DDR. The final step is to boot strap the processor to start executing the Linux OS (or bare-metal application).

SASB uses a physical unclonable function to generate the decryption key as a mechanism to eliminate the vulnerabilities associated with on-chip key storage. Key generation using PUFs requires an enrollment phase, which is carried out in a secure environment, i.e., before the system is deployed to the field. The enrollment process for SASB involves developing a set of challenges that are used by the PUF to generate the encryption/decryption key for AES.

During enrollment when the key is generated for the first time, the PUF accepts challenges, generates the key internally and transfers helper data off of the FPGA. As shown in Fig. 2, the challenges and helper data are stored in the external NVM unencrypted. The internally generated key is then used to encrypt the other components of the NVM by configuring AES in encryption mode. A special enrollment version of SASB is created to enable this process to be performed in a secure environment. The enrollment version performs encryption instead of decryption as is true for the in-field version but is otherwise identical.

The proposed system has the following security properties. First, the enrollment and regeneration processes proposed for SASB never reveal the key outside the FPGA. Therefore, physical, side-channel-based attacks are necessary in order to steal the key. We do not address side-channel attacks in this paper, but it is possible to design SASB with side-channel attack resistance using circuit countermeasures as proposed in [14]. Second, any type of tamper with the unencrypted helper data by an adversary will only prevent the key from being regenerated and a subsequent failure of boot process. Note that it is always possible to attack a system in this fashion, i.e., by tampering with the contents stored in the external NVM, independent of whether it is encrypted or not. A more significant concern relates to whether the helper data reveals information about the decryption key. As discussed in [10], the HELP PUF within SASB implements a helper data scheme that does not leak information about the key. Last, the proposed secure boot scheme stores an unencrypted version of the SASB bitstream and therefore, adversaries are free to change components of SASB and/or add additional functionality to the unused regions in the PL. This problem is addressed by using a PUF that can self-authenticate and detect tamper, as we discuss in the following section.

### 4.1 Attack Model

The primary attack model addressed by SASB is key theft. The adversary's goal is to add a key leakage channel via a hardware Trojan that would provide backdoor access to the key. In order to accomplish this, the unencrypted SASB bitstream needs to be reverse engineered. The attack process and options available to the adversary are illustrated in Fig. 3.

The attack modifications labeled $A_1$ in Fig. 3 involve changing wire and LUT configuration information within the SASB component of the bitstream as a means of providing back door access to the SASB key. The $A_2$ attack modifications illustrate the addition of a hardware Trojan that accomplishes the same goal. In both cases, the high valued target is the key register. The leakage channel is created by simply adding wires that route the key information to pads on the FPGA. The back door logic added by the adversary simply waits until the key is generated, which occurs in the 3rd step of the secure boot process as shown on left side of Fig. 2.

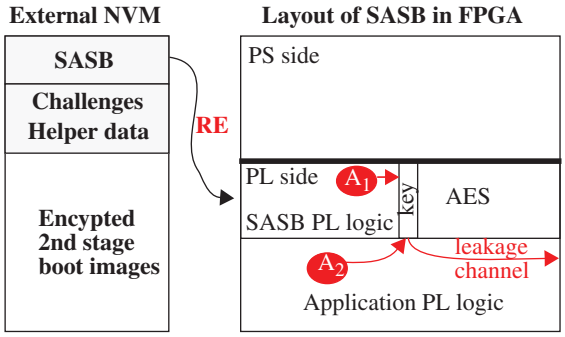


**Fig. 3. Attack model illustration.**

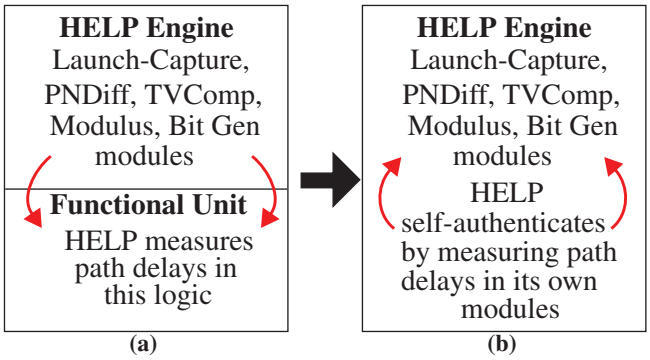


**Fig. 4. (a) Original HELP architecture, and (b) proposed changes.**

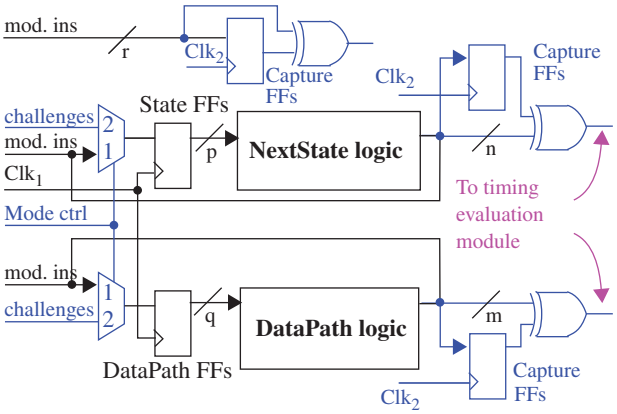


**Fig. 5. Dual mode architecture of the HELP modules.**

The goal of SASB is then to prevent a valid key from being read out through the back door. SASB implements a defense mechanism that detects tamper and scrambles the key if either of the modifications shown in Fig. 3 are attempted. The defense mechanism is based on measuring path delays within SASB at high resolution and then deriving the key from these measurements. Therefore, **correct regeneration of the key is dependent on the delays of a set of paths**. These paths were measured during enrollment to generate the key used to encrypt the second stage boot image. SASB re-measures the same paths when the system is booted in the field. The path delays are measured at a resolution that can detect any type of malicious change. The SASB key generation algorithm is constructed such that a change in any of these path delays causes a large number of bits in the key register to change. Therefore, the key read by the adversary is wrong and the system fails to boot. The details of this self-authentication operation are discussed in the next section.

## 5 Details of SASB

SASB leverages a PUF called the Hardware Embedded Delay PUF, first proposed in [8]. HELP measures path delays in arbitrarily-synthesized functional units, i.e., multipliers and cryptographic primitives, and uses the within-die variations that occur in these delays as a mechanism to generate a unique, device-dependent key. The HELP architecture as originally proposed is shown in Fig. 4(a). HELP utilizes the 'Functional Unit' as a dedicated source of entropy. The HELP Engine includes a set of modules as shown that measure path delays in the Functional Unit, and then uses these digitized delays in a key generation algorithm.

Fig. 4(b) shows the architecture proposed in this paper, which eliminates the 'Functional Unit' and instead **uses the implementation logic of the HELP engine itself as the source of entropy**. As we will show, all of the modules within HELP except for the Launch-Capture module can be configured to operate in one of two modes[1]. Mode 1 is the original mode, in which the module carries out its dedicated function as part of the HELP algorithm. Mode 2 is a special Launch-Capture (LC) mode, that allows the Launch-Capture module to apply 2-vector sequences to its inputs (challenges) and then measure the delays of a set of paths through the modules. The digitized representation of these path delays are stored in a BRAM and used later to generate the key when the modules are switched back to Mode 1.

The dual mode structure for a typical module is shown in Fig. 5. All the changes are implemented in an HDL, i.e., no hand-crafting of the wires and LUTs is necessary. The original HDL modules for HELP are written in a two-segment style to enable the second mode to be easily integrated. In two-segment style, the HDL statements that describe the storage elements, i.e., the State and DataPath registers (FFs), are described in a separate process block from the HDL that describes the NextState and DataPath combinational logic.

The elements shown in blue represent the changes required to provide two modes of operation for each of the HELP modules. The *Mode Ctrl* signal is used to switch between modes. All modules within HELP are converted into this type of self-authenticating structure except for those responsible for coordinating the launch-capture (LC) tests. All of the mode-configurable modules are tested simultaneously when configured in Mode 2 to ensure that the delays of paths between modules are also included in the key generation process. The module inputs (labeled *mod. ins*) in Fig. 5 are fanned-out to a dedicated set of Capture FFs to allow the inter-module paths to be timed.

The resource utilization of the original HELP architecture is estimated to be approximately 6000 LUTs (including the 3000 LUTs for the functional unit). The utilization with

1. Note that modifications to the LC module that change the behavior of the timing engine will prevent the key regeneration process from completing successfully. Therefore, this module does not require a self-authentication mode.

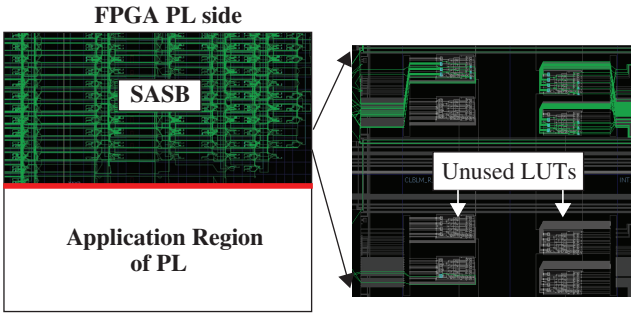


**Fig. 6. The SASB Isolation Region.**

the proposed changes is nearly equivalent because the overhead introduced by the dedicated functional unit (3000 LUTs) is eliminated in the SASB architecture, offsetting the overhead associated with the additional components shown in Fig. 5.

The HELP algorithm carries out a series of LC tests, called clock strobing. The 2-vector sequences (challenges) are delivered to the State and Datapath FFs by adding MUXs as shown on the left side of Fig. 5. Therefore, these registers also serve as the Launch FFs for Mode 2 of operation. A launch is carried out by putting the first vector, $V_1$, into these registers. The second vector, $V_2$, is then applied to the MUX inputs. On the next rising edge of $Clk_1$, transitions are launched into the combinational logic blocks. The delay of the paths are timed by capturing transitions that occur on the outputs of the combinational logic blocks by a set of Capture FFs. The Capture FFs are driven by a second clock, $Clk_2$, that is a phase shifted version of $V_1$. A digital clock manager is used to generate $Clk_2$.

Each of the challenges are applied multiple times. For each LC test, the phase shift of $Clk_2$ is incremented forward with respect to $Clk_1$ by a small $\Delta t$ (approx. 18 ps using the DCM in a Xilinx Zynq FPGA). Each of the paths driving the *n* and *m* outputs which have transitions will, for one of the LC tests, succeed in propagating its transition to the corresponding Capture FF before $Clk_2$ is asserted. When this occurs, the XOR gate monitoring the output becomes 0. The first occurrence of a 0 in the repeated sequence of LC tests applied causes the controlling LC module to store the current value of the phase shift as the digitized delay for the path. The XOR == 0 event occurs at different phase shifts for each of the paths so LC testing continues with larger and larger phase shifts until all paths are timed. The digitized path delays are stored in a BRAM for processing later by the HELP algorithm in Mode 1.

The challenges are designed in advance to provide complete coverage, i.e., all LUTs are tested using at least one delay test. Therefore, any changes to the logic functions implemented within the LUTs, and any wiring changes to the inputs or outputs of the LUTs will change the delay characteristics of the measured paths. Adversaries can also snoop on data values that are produced during key regeneration (Mode 1) as a mechanism to steal the key. This can be achieved by adding fanout branches to the existing wires. Unfortunately, the corresponding changes in the path delays are too small to be detected by the proposed self-authentication method. We propose a separate mechanism for dealing with fanout branch insertion in the following sections.

### 5.1 SASB Isolation Region

SASB is designed to minimize its usage of PL resources as a means of maximizing the resources available for an application bitstream. Moreover, SASB is configured into a Xilinx *pblock* as shown in Fig. 6 (not drawn to scale). We refer to this region as an isolation region because the LUTs and switch boxes within this region are used either by the SASB modules or are configured into dummy paths and timed in the same fashion as described for Mode 2 operation of the SASB modules. Note that the isolation region proposed here is fundamentally different than the *fence* construct proposed by Xilinx [1]. Unlike the fence, the SASB isolation region implements an active self-authentication mechanism against tamper.

The unused LUTs are identified using a *tcl* script once the synthesis completes [13]. Several unused LUTs are shown on the right side of Fig. 6. Dummy paths are constructed by creating a path through these LUTs with the endpoints connected to a Launch and Capture FF. The objective is to create dependencies between key generation and all of the unused LUT resources within the isolation region as a mechanism to prevent adversaries from using these LUTs to construct a key snooping Trojan.

The tactic of stringing together the unused LUTs into structural paths does not address Trojans that create paths from, e.g., the key register, directly to the FPGA I/Os. LUT resources are not required to create routes. Instead, the switch boxes must be protected. This can be accomplished by preventing adversaries from creating fanouts on wires connected to the registers that store the regenerated key (other wires that process sensitive information related to the key can also be treated in this fashion). The basic idea is to route ***fanout-blocking*** paths through switch boxes used to route key information. The switch boxes provide the only opportunity for adversaries to create fanout to these wires. The fanout-blocking paths effectively use all of the available fanout connections through the switch that can be connected to the key register.

An example of a fanout-blocking-path is shown in Fig. 7. A key bit is generated and stored in the LUT shown on the right. The wire shown in yellow routes from the key register to another component in the isolation region, e.g., an input to AES. The point at which the yellow wire enters the switch box on the right represents a vulnerability. The switch box allows fanout connections to be made to wires passing through the switch box [9]. The white wire shows an example of a fanout connection that can be made to the yellow wire. In Xilinx Zynq-7000 FPGAs, the number of possible fanouts (PIPs) permitted in a switch box node is 32. We identify these fanout branch options within the switch boxes used to route the key register bits and add fanout-blocking paths, such as the one shown in magenta. A launch and capture FF are added to the endpoints of the fanout-blocking paths (the magenta wire is driven from one as shown on the right in the figure) to allow these paths to be timed, and participate in the key generation process. Adversaries who attempt to re-route the fanout-blocking path to gain access to the fanout connection will change the delay of the path, which will result in

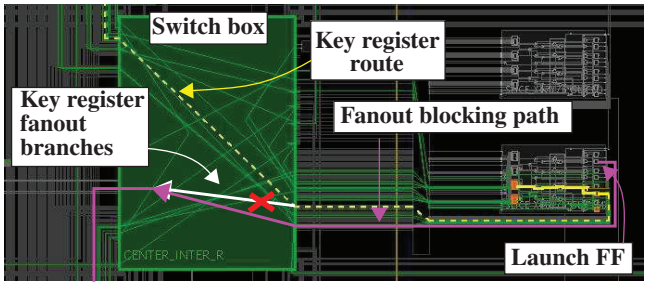


**Fig. 7. Fanout branches within a Xilinx switch box.**

key generation failure. Therefore, the key bit snooped is purposely corrupted by SASB because of the change in the path delay.

### 5.2 Sensitivity Analysis of Self-Authentication Technique

The security of the proposed scheme is rooted in the ability to detect changes to the routing of existing wires within the SASB modules and to the fanout-blocking paths. In this section, we present results that show the change in delay that results from minimal changes to the routing configuration of a path. The data is obtained from measurements on a Xilinx Zynq 7020 using the timing engine implemented within SASB.

Vivado implementation view is used to create manually -routed paths through two switch boxes between two adjacent slices. A second configuration is created that adds one additional switch box to the path, to model an adversarial attack that attempts to re-route a fanout-blocking path represented by the first configuration. The delay of the first configuration is 558 ps while the second configuration adds 72 ps. The increased delay in this 'hardest-to-detect' attack model is large enough to cause a bit flip error in the HELP bitstring generation algorithm.

### 5.3 Blanking Bitstream Countermeasure

As indicated, the adversary can place a key snooping Trojan circuit into the unused Application Region of the PL. An effective countermeasure to preventing this is to enable SASB to write a blanking bitstream into the Application Region using the Xilinx ICAP interface **before the key is generated** [11][12]. This would destroy the Trojan before it could be activated. SASB includes a module that performs a partial dynamic reconfiguration on the Application Region of PL from Fig. 6 using a state machine that automatically generates blanking data required by the ICAP interface. Given that SASB is unencrypted, the adversary might attempt to disable this state machine or change its functionality. As a countermeasure, SASB also self-authenticates the blanking bitstream state machine as part of the key generation process. This technique can also be used to eliminate the requirements for fanout-blocking paths in the isolation region. By moving the SASB module away from the edges of the design to allow the blanking region to surround SASB, any attempt to create routes that leak the key register to I/O would be eliminated by the blanking bitstream operation.

### 5.4 SASB Key Generation with Avalanche Effect

In order to prevent incremental attacks, the key generation process creates dependencies between the bitstrings generated by HELP and the AES key. Therefore, any single bit flip that occurs in the HELP bitstrings because of tamper to a path will propagate to multiple key bits. The avalanche effect is a well know property of secure hashing algorithms such as the SHA-3. Therefore, the bitstrings generated by the HELP algorithm, as components of SASB are self-authenticated, are used as input to a SHA-3 implementation embedded within the SASB bitstream. The digest is then used as the AES key to decrypt the second stage boot loader images.

Note that the avalanche effect behavior of SHA-3 does not increase the reliability requirements of the PUF. This is true because key regeneration has zero tolerance to bit flip errors, and is independent of the hashing operation. However, key regeneration is now being performed over a much larger sequence of bits and therefore, the reliability requirements of the HELP algorithm are increased by a factor proportional to the compression performed by the hashing operation. The HELP algorithm includes several reliability-enhancing techniques and corresponding parameters that can be tuned to increase the reliability of HELP's bitstring regeneration process to achieve a specific requirement for key regeneration.

## 6 Conclusions

A PUF-based secure boot technique called SASB is proposed that is designed to self-authenticate as a mechanism to detect tamper. An unencrypted version of SASB, which is stored in an external NVM, is loaded by the first stage boot loader. The PUF within SASB regenerates a decryption key by measuring variations in path delays that occur within the SASB modules. This self-authentication process detects tamper attacks that modify the LUTs or routing within the SASB modules in an attempt to create a leakage channel for the key. The conceptual design of SASB is described and experimental results presented that investigate the sensitivity of the timing measurements to changes in the structural characteristics of paths, as a illustration that such changes can be detected and will result in failure to boot the system.

## 7 References

[1] S. M. Trimberger, J. J. Moore, "FPGA Security: Motivations, Features, and Applications", Invited paper, *Proceedings of the IEEE,* Vol. 102, No. 8, 2014, pp. 1248-1265.
[2] S. Skorobogatov, "Flash Memory 'Bumping' Attacks", *Cryptographic Hardware and Embedded Systems*, 2010.
[3] P. Swierczynski, M. Fyrbiak, P. Koppe, C. Paar, "FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives", *Computer-Aided Design of Integrated Circuits and Systems*, 2015.
[4] P. Swierczynski, G. T. Becker, A. Moradia and C. Paar, "Bitstream Fault Injections (BiFI) - Automated Fault Attacks against SRAM-based FPGAs", *Transactions on Computers*, 2018.
[5] P. Swierczynski, M. Fyrbiak, P. Koppe, et al., "Interdiction in Practice—Hardware Trojan Against a High-Security USB Flash Drive", *J Cryptogr Eng*, 2017.
[6] D. Konopinski and A. Kenyon, "Data recovery from damaged electronic memory devices", *London Communications Symposium*, 2009.
[7] http://chipdesignmag.com/display.php?articleId=5045
[8] J. Aarestad, P. Ortiz, D. Acharyya, J. Plusquellic, HELP: A Hardware-Embedded Delay-Based PUF, *Design and Test of Computers*, 2013.
[9] https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
[10] W. Che, M. Martin, G. Pocklassery, V. K. Kajuluri, F. Saqib, and J. Plusquellic, "A Privacy-Preserving, Mutual PUF-Based Authentication Protocol", *Cryptography*, 2017.
[11] https://forums.xilinx.com/t5/Embedded-Processor-System-Design/How-to-use-PCAP-to-config-the-PL-in-zynq/td-p/280230
[12] https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf
[13] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug904-vivado-implementation.pdf
[14] K.Tiri, I.Verbauwhede, "Charge Recycling Sense Amplifier Based Logic: Securing Low Power Security ICs Against DPA", *Solid-State Circuits Conference*, 2004.