Secure Boot Introduction

Embedded system security can be partitioned into two categories:

- Security issues associated with the **design**
- Security issues associated with the system and application **data** once the design is instantiated (or programmed for FPGAs) and the system is booted

Design security can be sub-divided into:

- Bootstrap: Techniques for ensuring that authentic BootROM is run or the authentic design is programmed onto the FPGA
- Trojan insertion: Techniques for preventing or detecting adversarial insertions of malicious functions to a design
- IP protection: Techniques to prevent adversaries from reverse engineering the design and/or leveraging the design in their own applications
- IC overbuilding: Methods for preventing adversaries from building additional copies of an authentic design, or coping and using legitimate bitstreams illegally

The first two of these relate to **Secure Boot**

> Methods that guarantee that the system boots with an authorized FPGA bitstream and/or BootROM code establish the 'root of trust' in the system

Secure Boot Introduction

A running system booted in a trusted fashion can then be used to ensure **data** security

Data security can be sub-divided into:

- Protecting stored data and user application keys
- Protecting the processing of that data
- Protecting the transmission and reception of data with other parties

Methods here include

- Tamper detection that destroys (zeroizes) all sensitive data
- Encryption and authentication mechanisms
- Secure key storage mechanisms via PUFs
- Network hardware firewalls
- Differential power analysis countermeasures

Of course, none of these can be ensured unless the system boots in a trusted and secure fashion

Secure Boot Introduction

The focus of our discussion will be on secure boot of FPGAs

We will examine the methods provided by commercial FPGA vendors, in particular, Xilinx, for achieving this

And then we will discuss a novel technique that we are proposing based on the HELP PUF

In Xilinx FPGAs, the root of trust is the stored key

Keys can be stored in Battery Backed RAMs (**BBRAM**) or using **eFUSE**

The drawbacks of these on-chip digital storage mechanisms include

- BBRAM require a battery to be installed on the system board and therefore increase system cost
- The batteries for BBRAM also have a limited lifetime and therefore complicate system maintenance
- eFUSE is one-time-programmable (OTP) and therefore reduce flexibility in key management
- eFUSE keys can be read-out using, e.g., scanning electron microscopes (SEM)

Xilinx Secure Boot Process

The BBRAM or eFUSE keys are used as the root of trust in the Xilinx secure boot process

- In a secure facility, the Xilinx CAD tools can be used to encrypt the bitstream using a randomly generated or user-specified key

- The decryption key is loaded via JTAG at a secure facility into the eFUSE or BBRAM

- The in-field secure boot process first determines if the external bitstream includes an encrypted-bitstream indicator

  If so, the on-chip 256-bit AES engine decrypts the bitstream using cipher block chaining (CBC) mode of AES along with the eFUSE or BBRAM key

  CBC mode XORs the previous block ciphertext with the next block plaintext before encrypting the current block (decryption reverses this process)

  This forces different ciphertexts for replicated components in the plaintext

Xilinx Secure Boot Process

- Authentication is used to ensure data integrity of the bitstream using SHA-256 where a 256-bit *keyed MAC* (**HMAC**) is computed for the bitstream

  The HMAC is designed to prevent bit-flip attacks and other types of fault injection attacks

  Therefore, the HMAC authenticates the origin of the bitstream and detects any type of tamper

  The HMAC of the unencrypted bitstream is computed in a secure facility and embedded with the key in the bitstream, which is then 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, data integrity and authentication

It detects tamper and attempts to program FPGA with a non-authentic bitstream

Xilinx SoC Secure Boot Process

Xilinx FPGA SoCs, e.g., Zynq series, use an asymmetric (public-private) authentication (digital signature) scheme in the secure boot process

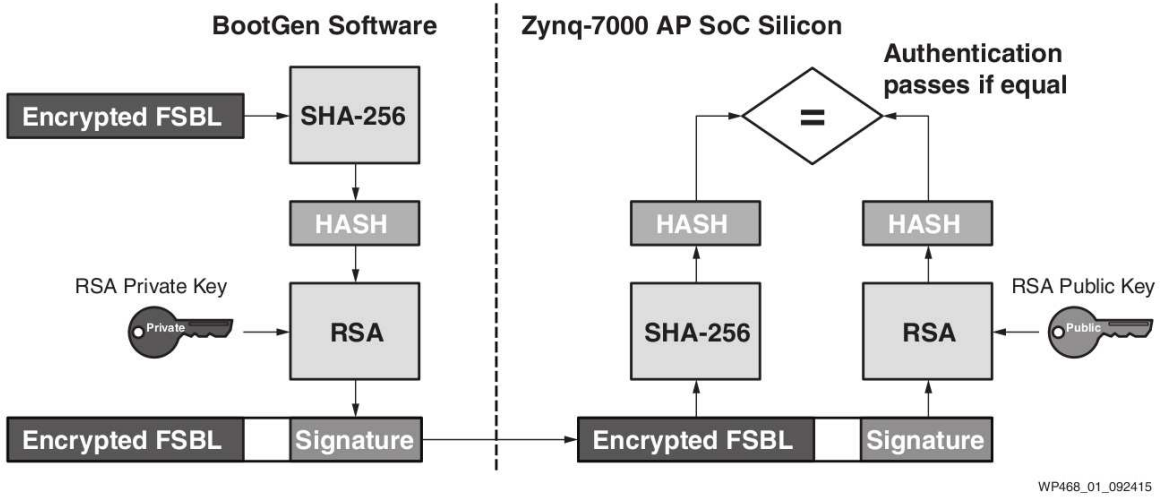


Figure 1: **Asymmetric Authentication Process**

Leveraging Asymmetric Authenticationto Enhance Security-Critical Applications Using Zynq-7000 All Programmable SoCs,WP468 (v1.0) October 20, 2015

Here, we see **bootgen** computes a SHA-256 hash of the encrypted first stage boot loader (**FSBL**) and a *digital signature* is then computed using the RSA private key

Signature verification is carried out by the Zynq chip using the public key to recover the hash, which is compared with a locally computed hash of the encrypted FSBL

Xilinx SoC Secure Boot Process

The first stage boot loader (FSBL) is authenticated as shown BEFORE it is decrypted and executed by the PS-side

If authentication succeeds, the FSBL is decrypted by a PL-side AES engine using a key stored in the BBRAM or eFUSE

RSA-2048 signature verification algorithm resides in the PS-side BootROM, which is a mask-programmed, hardwired, immutable memory

Neither the private or public keys are stored on the FPGA

Instead, a 256-bit hash of the public key is programmed into the eFUSE array

The **FSBL** then becomes the **root of trust** in the boot process

PS-side images and PL configurations can then be loaded by the FSBL

The user must include decryption and authentication functions in the FSBL to ensure these subsequent components of the boot process are secure

Xilinx Secure Boot Process

Secure boot requires the boot process to begin with a root of trust, and then carry out authentication in each of the subsequent stages

As indicated above, Xilinx FPGA SoCs use 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 eFUSE embedded key

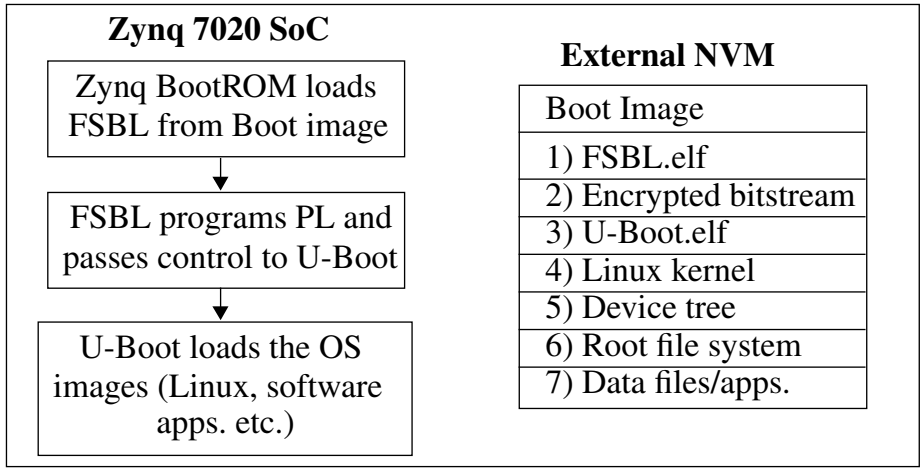
Although the Xilinx FPGA SoC root of trust begins with the RSA authenticated FSBL, which does not use an embedded key, decryption of the FSBL does

Moreover, the Xilinx non-SoC PL-side boots, as discussed earlier, use eFUSE and BBRAM for bitstream decryption

In either case, the **root of trust cannot be expanded** to include PS-side images and/or PL configuration data without keeping the embedded key confidential

Xilinx Boot Process

Let's examine the underlying steps of the Xilinx boot process and then look at an alternative self-authenticating PUF-based solution



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 includes a bare-metal application, or the Linux OS, embedded software applications and data files

Self-Authenticated Secure Boot (SASB) Process

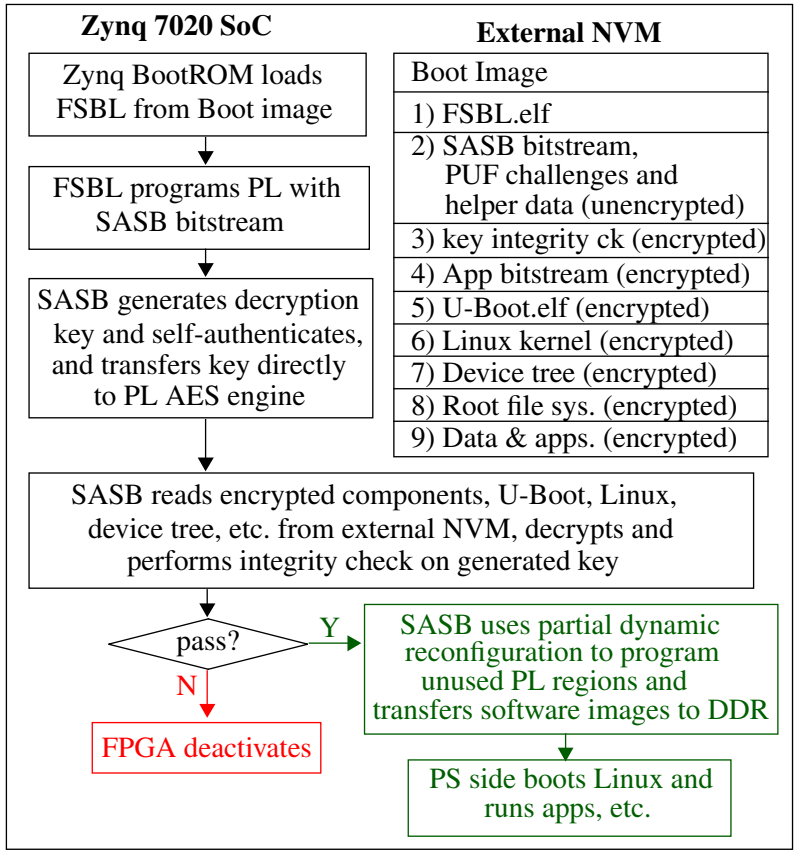
The Self-Authenticated Secure Boot (SASB) boot process **does not** use any of the security features provided by Xilinx, i.e., it is self-contained and self-authenticaing

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 bitstream

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

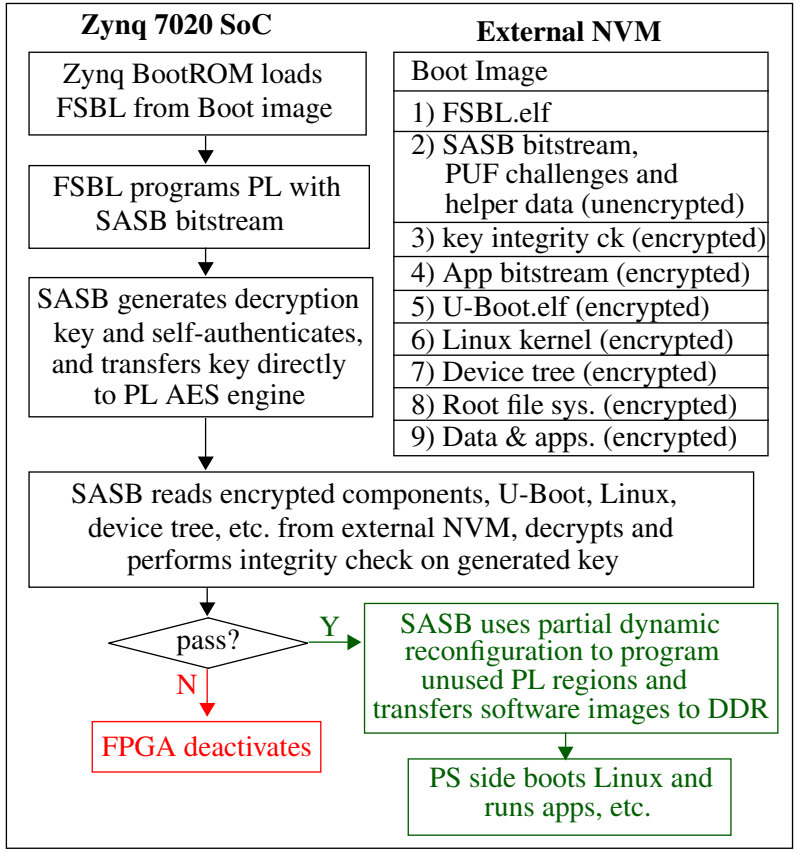
Self-Authenticated Secure Boot (SASB) Process

SASB reads the encrypted second stage boot image components labeled as components 3 through 9 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)

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



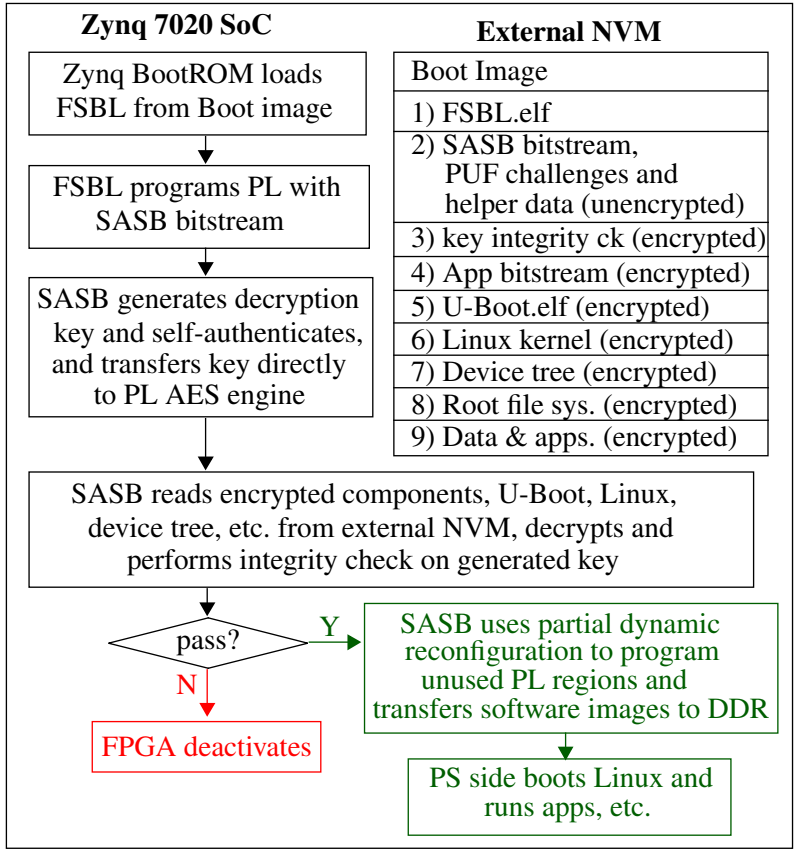
Self-Authenticated Secure Boot (SASB) Process

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

Self-Authenticated Secure Boot (SASB) Process

SASB uses the HELP PUF to generate the decryption key as a mechanism to eliminate the vulnerabilities associated with on-chip key storage

Key generation using PUFs starts with an enrollment phase carried out in a secure environment

Challenges are applied to generate the encryption key for encrypting the 2nd stage boot images

A special enrollment version of SASB generates the key internally and transfers helper data off of the FPGA

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

The enrollment version performs encryption while the in-field version performs decryption, but the two versions are otherwise identical

Security Properties of SASB

The proposed system has the following security properties

- The enrollment and regeneration processes **never reveal the key** outside the FPGA, requiring the adversary to use physical, side-channel-based attacks to steal the key

- 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 tamper with the contents stored in the external NVM, independent of whether it is encrypted or not

- The HELP PUF discussed earlier implements a helper data scheme that does not leak information about the key

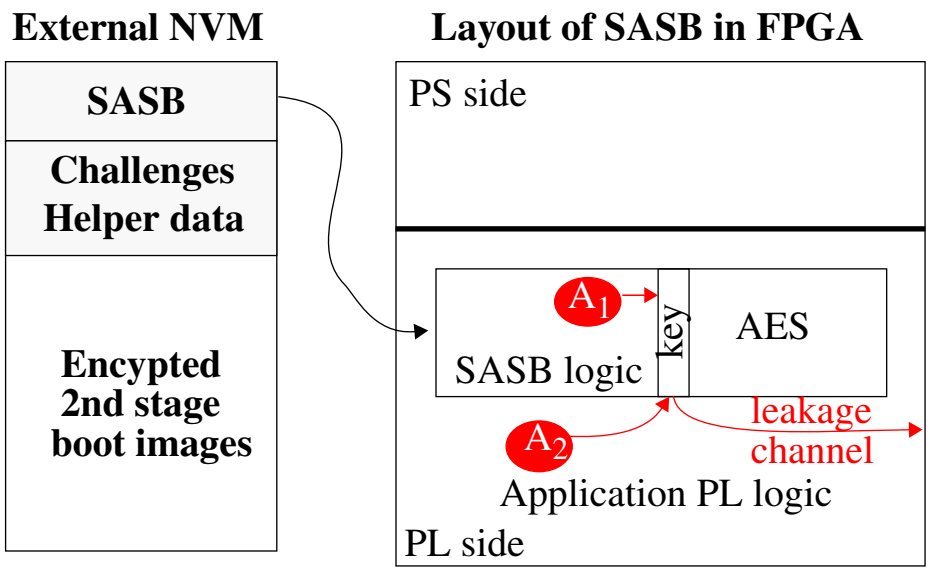
- The HELP PUF to designed to self-authenticate itself, thereby detecting any type of tamper with unencrypted version of the SASB bitstream

SASB 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 first needs to be reverse engineered



The attack modifications labeled ***A1*** involve changing wire and LUT configuration information within SASB

The ***A2*** attack modifications illustrate the addition of a hardware Trojan outside of SASB

In both scenarios, the objective is to leak the key to I/O pads

SASB Attack Model

The back door logic added by the adversary could simply wait until the key is generated, which occurs in the 3rd step of the secure boot process

SASB implements defense mechanisms that detects tamper and scrambles the key if ***A1*** is attempted, and deletes the hardware Trojan when ***A2*** is attempted

The defense mechanisms are based on measuring path delays within SASB at high resolution and then deriving the key from these measurements

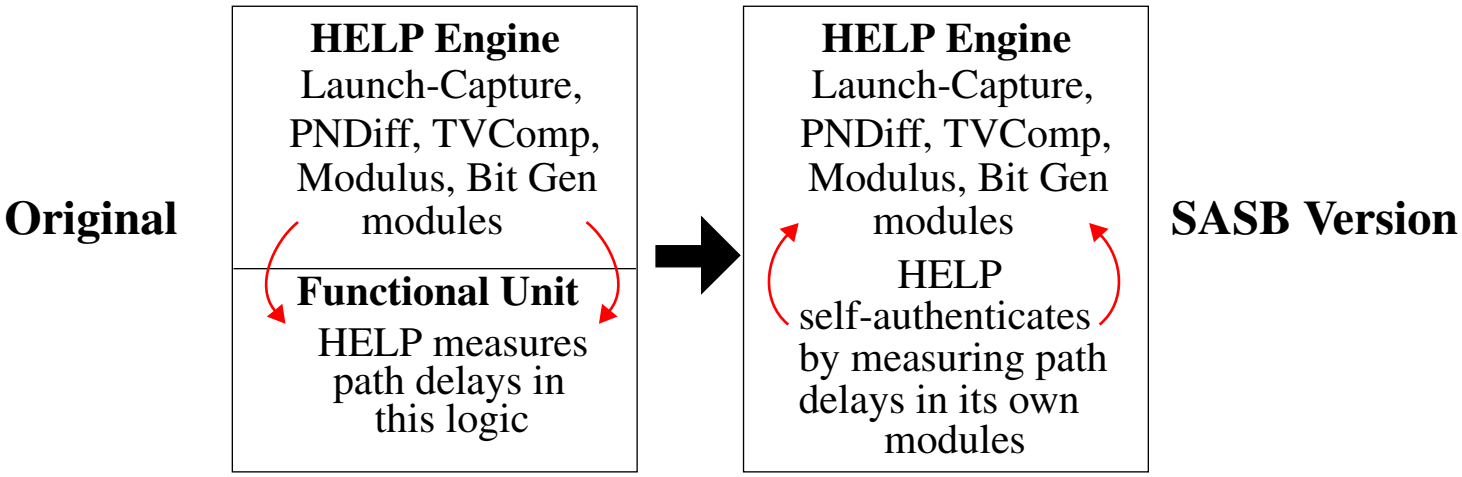
Therefore, correct regeneration of the key is dependent on the delays of a set of paths

The SASB key generation algorithm is constructed such that a change in any of the path delays b/c of tamper 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

SASB Architecture

The original and newly proposed architecture for HELP



HELP leverages variations in path delays in arbitrarily-synthesized functional units, i.e., cryptographic primitives to generate a unique key (see PUF screencasts)

The HELP Engine includes a set of modules that measure path delays in the Functional Unit, and then uses these digitized delays in a key generation algorithm

The SASB architecture eliminates the 'Functional Unit' and instead **uses the implementation logic of the HELP engine itself as the source of entropy**

SASB Architecture

All of the modules within HELP except for the Launch-Capture module are configured to operate in one of two modes

- **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 mode, that allows the Launch-Capture module to apply 2-vector sequences to its inputs and then measure the delays 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

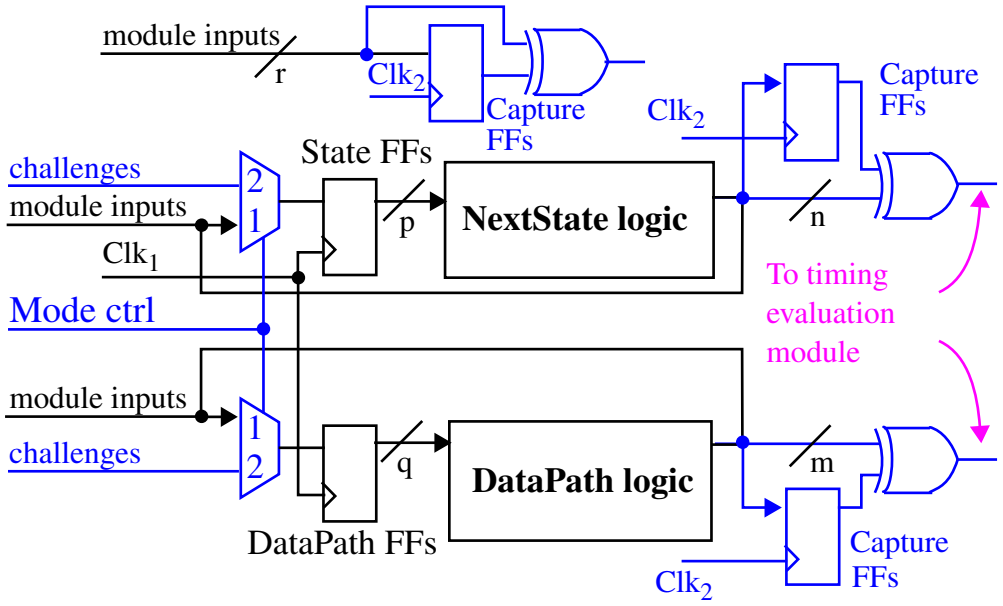
All the changes are implemented in an HDL, i.e., no hand-crafting of the wires and LUTs is necessary

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, State and DataPath registers (FFs) are described in a separate process block from the NextState and DataPath combinational logic

SASB Architecture

The dual mode structure for a typical SASB module



The elements shown in blue represent the changes required to provide two modes of operation for each of the SASB modules

The 2-vector sequences (challenges) are delivered to the State and Datapath FFs by adding MUXs as shown on the left side

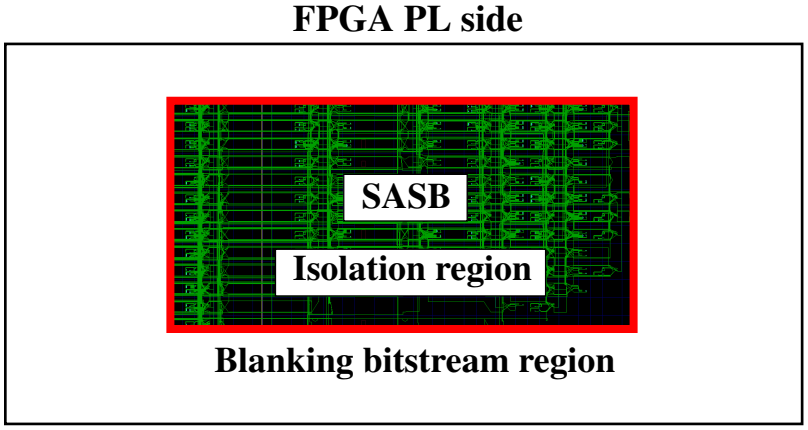
HELP uses a *clock strobing* technique to time the paths (see PUF screencasts)

SASB Architecture

Note that any tamper to the logic functions implemented within the LUTs will change the **functional behavior** and result in missing or extra timing values

As mentioned, adversaries can also snoop during key regeneration in Mode 1 as a mechanism to leak the key **by adding fanout branches**

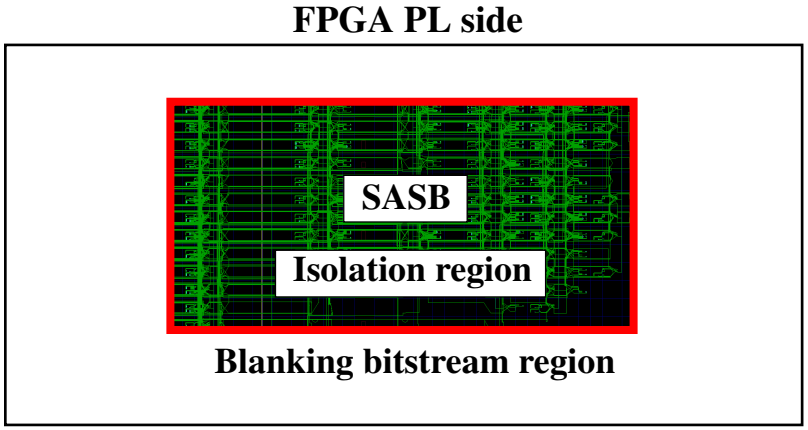
As a countermeasure, SASB is configured into an *isolation region* (a pblock), which is surrounded by the *blanking bitstream* region



SASB writes a blanking bitstream into this region using the ICAP interface before the key is generated destroying Trojans and any switch box routing information

SASB Architecture

SASB includes a module that performs partial dynamic reconfiguration on the blanking bitstream region



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

Other proposed countermeasures include wiring unused LUTs together within the isolation region and creating *fanout-blocking* paths through the switch boxes

BulletProof Architecture

We developed a variation of SASB called Bullet-Proof Boot for FPGAs (**Bullet-Proof**)

BulletProof derives challenges for the HELP PUF using the FPGA configuration data read directly from the ICAP interface

Since the FPGA is programmed with the unencrypted BulletProof bitstream, this also represents a second form of self-authentication

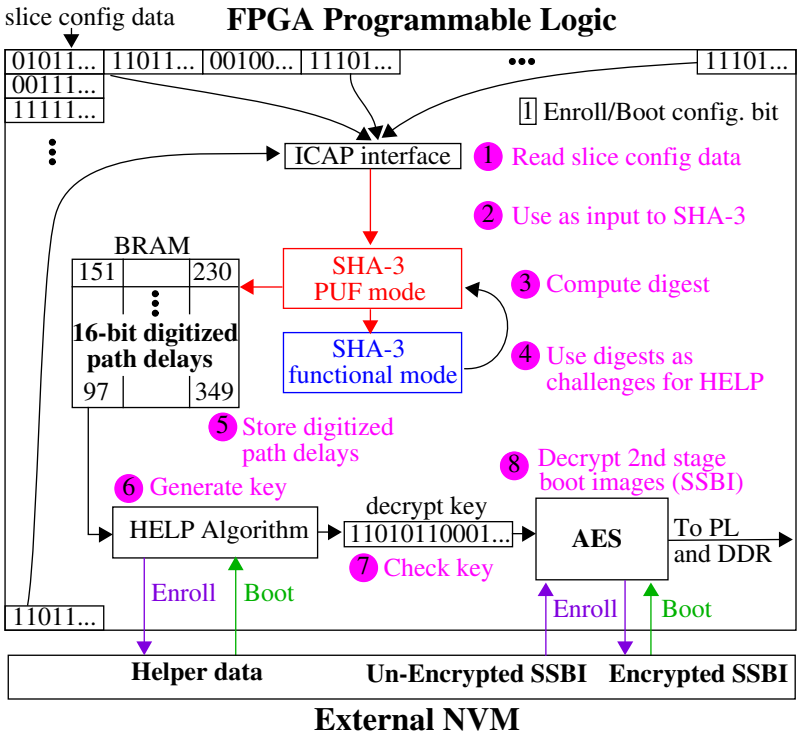
The source of entropy of the HELP PUF is an implementation of the SHA-3 algorithm, in contrast to the HELP modules themselves as was true for SASB

The bitstream configuration data is hashed using SHA-3 configured in Mode 1 (functional mode)

Periodically, the current state of the SHA-3 hash is used as a challenge to SHA-3 configured in Mode 2 (PUF mode) to generate timing data for key generation

BulletProof Architecture

The configuration data within the PL-side of the FPGA is shown overlaid on top of the BulletProof flow diagram

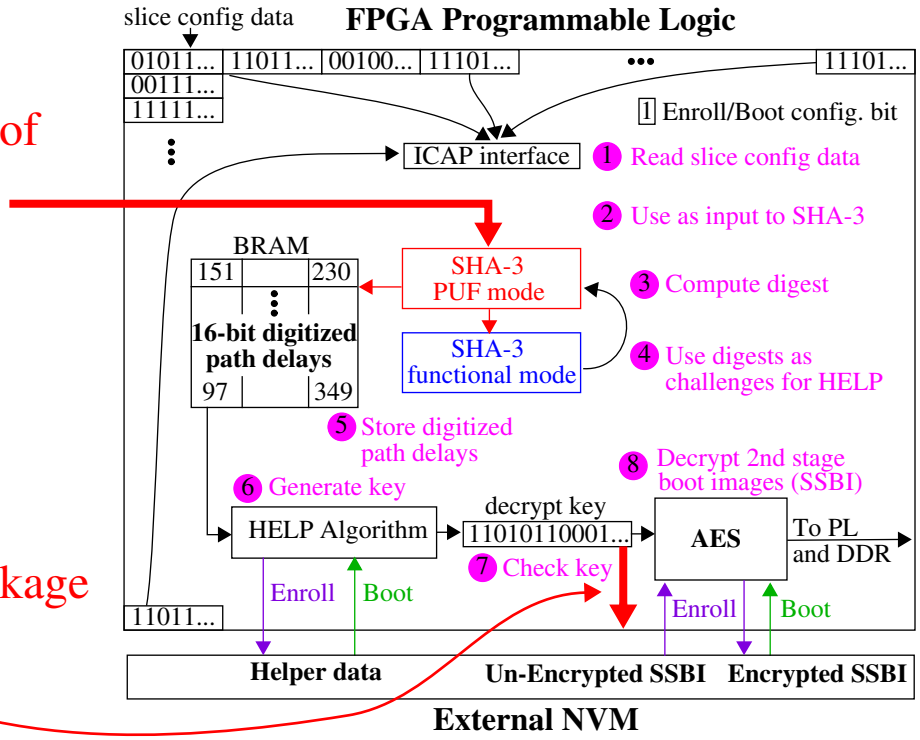


The SHA-3 blocks are shown as two separate blocks but are in fact one block

The BulletProof architecture is designed such that challenges are launched directly from the ICAP interface register to prevent a specific type of RE attack

BulletProof Architecture

The paths between the ICAP and SHA-3 are timed because of the following reverse engineering attack scenario



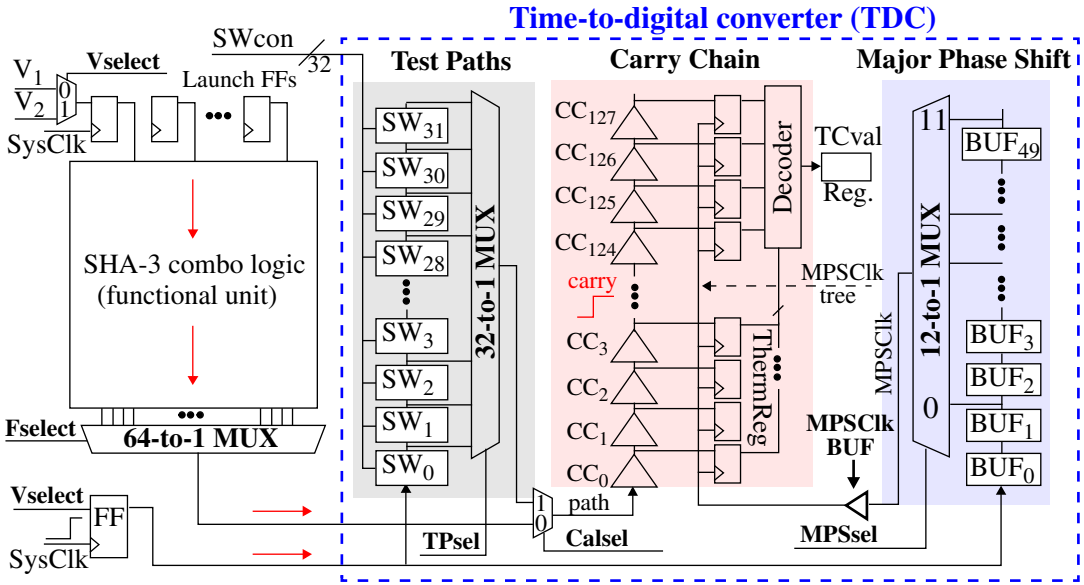
We must guarantee that the configuration data used as input to the SHA-3 originates from the ICAP interface

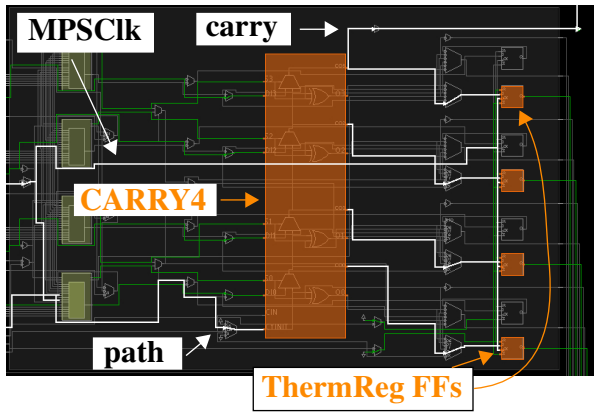
Otherwise, the adversary can create a route as shown and then change the on-chip version of BulletProof to leak the key off-chip

Time-to-Digital Converter Alternative to Xilinx MMCM

The screencasts on the HELP PUF discussed the *clock strobing* method that we use to time paths

An alternative is to use the high-speed carry chains on the FPGA in a time-to-digital converter (TDC) configuration





The TDC timing engine replaces the Xilinx MMCM, and when used with a ring oscillator as the clock source, prevents attacks that attempt to stop the clock