Implementation of a PUF Based Secure Boot Method for FPGAs

Summer Undergraduate Research in Engineering

Department of Electrical and Computer Engineering

McGill University

Xiangyun Wang

Student ID: 260771591

Supervised by: Prof. Zeljko Zilic

Secure Boot Overview

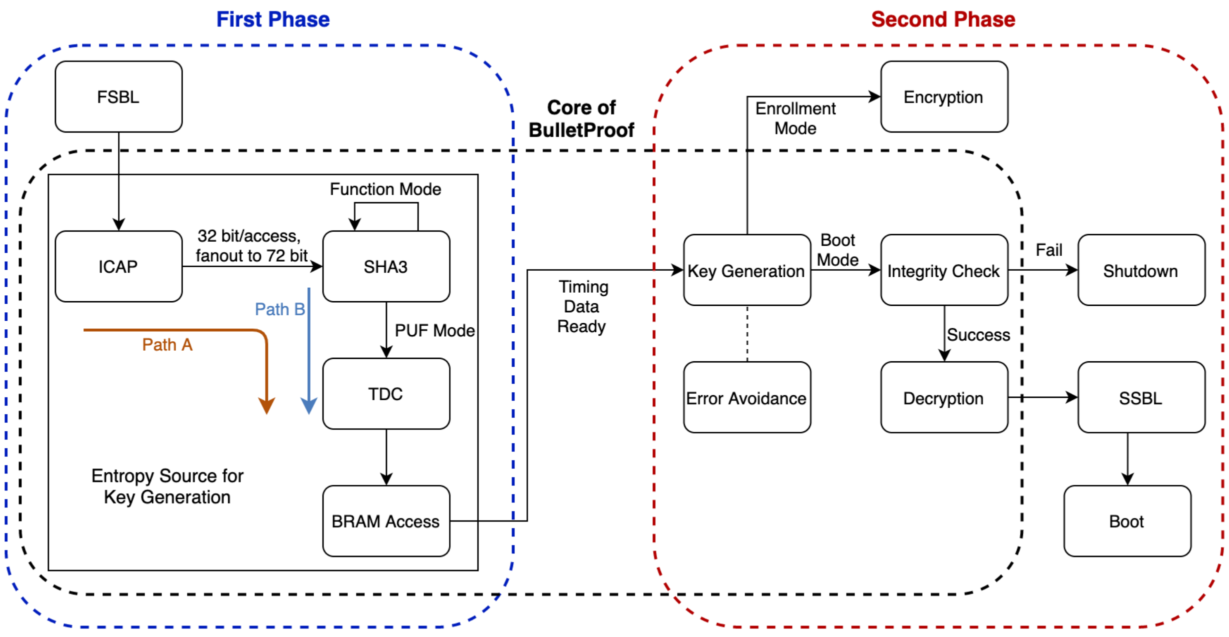


Figure 1 - BulletProof Secure Boot Flowchart

The flowchart above (Figure 1) shows the simplified process of the BulletProof secure boot.

First, the first stage boot loader (FSBL) programs the Programmable Logic (PL) side with the unencrypted BulletProof bitstream, shown as the “Core of BulletProof” in Figure 1, and hands over control to the BulletProof.

The BulletProof then reads the configuration data of the PL side using the ICAP interface, and generates timing data and digests using SHA3. The SHA3 first generates digests in function mode. Then, it switches to the PUF mode to generate timing data by applying the current digest into the SHA3 combinational logic. The generated timing data is stored in an on-chip block RAM (BRAM).

There are two paths, ‘A’ and ‘B’, that need to be timed, as shown in Figure 1. Path ‘A’ starts from the ICAP interface, and path ‘B’ starts from the SHA3 combinational logic. Path ‘B’ is used for the normal timing data generation, and path ‘A’ is used to prevent the adversarial modifications on the path between ICAP interface and SHA3.

When all the timing data is ready, the key will be generated using the timing data and helper data. Helper data contains information about unstable paths that could generate inconsistent timing data, so that during the key generation, only consistent data will be used, to avoid key generation error. After the key generation, if the BulletProof is in enrollment mode, then it will encrypt all other data (SSBL, OS image file, etc.). If the BulletProof is in boot mode, it then runs an integrity check for checking key correctness. There should be a small portion of unencrypted bitstreams in the external NVM, which will be used to compare with the output of the integrity check. If the output matched, then BulletProof will finish doing the decryption; otherwise, the boot process will be terminated.

After the decryption, the PL side will be programmed with the decrypted bit stream using dynamic partial reconfiguration, which is a function provided by Xilinx, and the normal boot process will continue.

Implementation

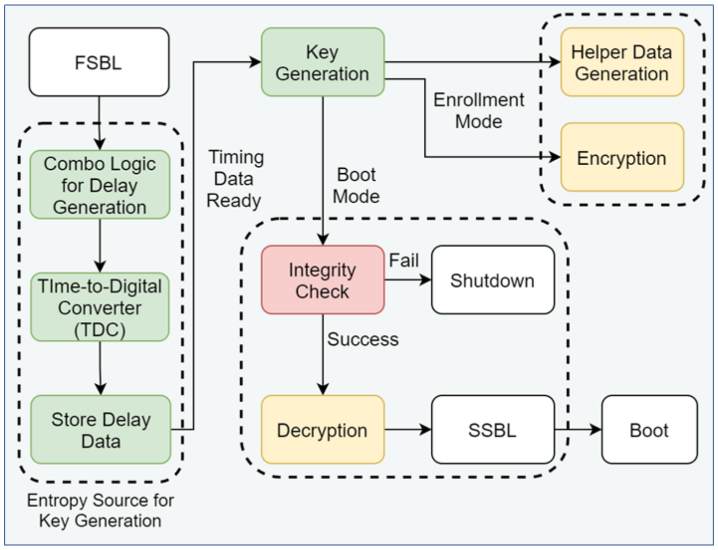


Figure 2 - Secure Boot Implementation Flowchart

The implementation is divided into 4 parts, which are the entropy source, the key generation, the enrollment mode, and the boot mode. As shown in the figure above, the blocks labelled in green are finished, the blocks in yellow and red are the parts that have been researched but not yet implemented, and the blocks in white are not yet being considered. For the parts that have been researched, all the useful resources are in the appendix.

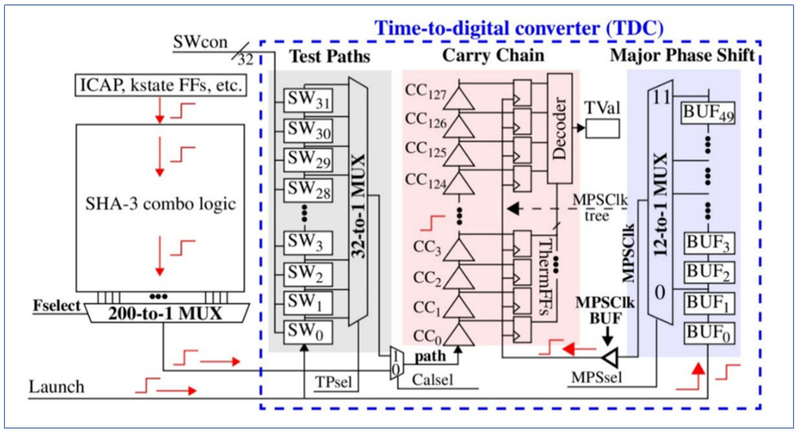


Figure 3 - Entropy Source Design[1]

The above figure (Figure 3) shows the detailed design of the entropy source for key generation. To the left is a Keccak-f[200] SHA3 combo logic that is used for delay generation. Bitstreams from ICAP and Flip-Flops will be fed into the SHA3, and the path that needs to be timed will be selected using a 200-to-1 multiplexer. To the right is the Time-to-digital converter (TDC), which consists of Test Paths, Carry Chain and Major Phase Shift, and will be used to time the signal propagation delay.

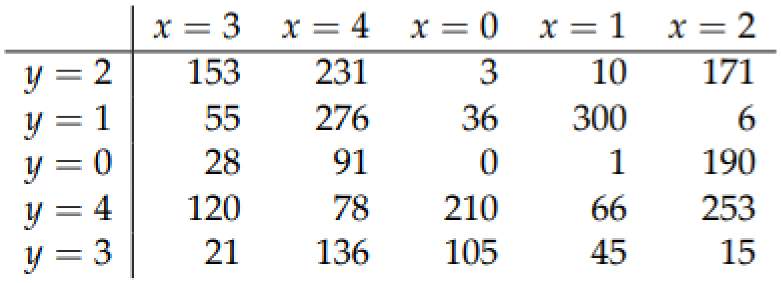
SHA3

According to [1], due to the hardware constraint of the ICAP interface, the Keccak-f [200] version of SHA3 should be used. The major limitation is the 32-bit readout of the ICAP interface, which limits the size of the input message for the SHA3. The ICAP interface will each time read out 32 bits of the hardware configuration, which will later be fanned out to 72 bits, so that it can be inserted to the SHA3.

Since the SHA3 function that will be used is not a standard Keccak-f[1600] version, the new Cyclic Shift Offset and Rotation Constant need to be determined. Both of them can be calculated using the algorithms provided by the Keccak Team, as shown in Figure 4 and 6, and the calculated results are shown in Figure 5 and Table 1.

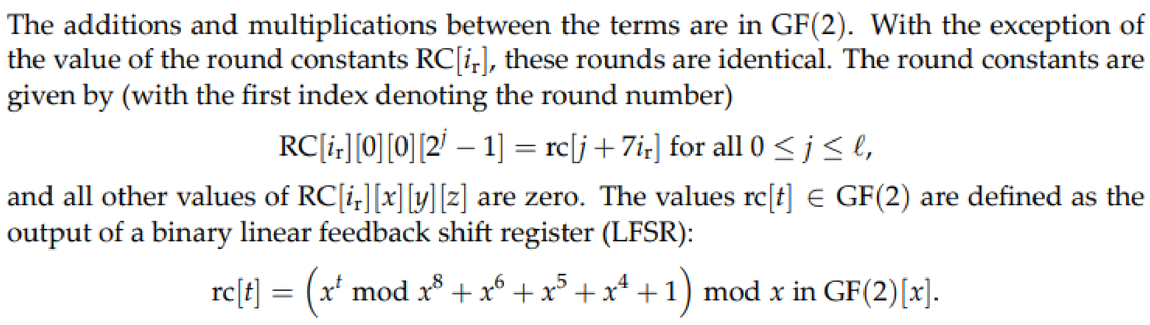
$$\begin{aligned}\rho: \quad a[x][y][z] \quad &\leftarrow a[x][y][z-(t+1)(t+2)/2], \\ &\text{with } t \text{ satisfying } 0 \le t < 24 \text{ and } \begin{pmatrix}0 & 1\\ 2 & 3\end{pmatrix}^t \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}x\\y\end{pmatrix} \text{ in } \mathrm{GF}(5)^{2\times 2}, \\ &\text{or } t=-1 \text{ if } x=y=0,\end{aligned}$$

Figure 4 - Formula for Cyclic Shift Offset[2]



	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	153	231	3	10	171
$y=1$	55	276	36	300	6
$y=0$	28	91	0	1	190
$y=4$	120	78	210	66	253
$y=3$	21	136	105	45	15

Figure 5 - Cyclic Shift Offset for Lane Size of 8[2]



The additions and multiplications between the terms are in GF(2). With the exception of the value of the round constants $\mathrm{RC}[i_r]$, these rounds are identical. The round constants are given by (with the first index denoting the round number)

$$\mathrm{RC}[i_r][0][0][2^j-1] = \mathrm{rc}[j+7i_r] \text{ for all } 0 \le j \le \ell,$$

and all other values of $\mathrm{RC}[i_r][x][y][z]$ are zero. The values $\mathrm{rc}[t] \in \mathrm{GF}(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$$\mathrm{rc}[t] = \left(x^t \bmod x^8+x^6+x^5+x^4+1\right) \bmod x \text{ in } \mathrm{GF}(2)[x].$$

Figure 6 - Formula for Round Constant[2]

Table 1 - Round Constant for Lane Size of 8

RC[0]	0x01	RC[6]	0x81	RC[12]	0x8B
RC[1]	0x82	RC[7]	0x09	RC[13]	0x8B
RC[2]	0x8A	RC[8]	0x8A	RC[14]	0x89
RC[3]	0x00	RC[9]	0x88	RC[15]	0x03
RC[4]	0x8B	RC[10]	0x09	RC[16]	0x02
RC[5]	0x01	RC[11]	0x0A	RC[17]	0x80

For Keccak-f[200] implementations, the state size should be 200 bits. The state block should have a shape of 3-D bit array, with size of 5*5*8 = 200 bits, where the 5*5 = 25 represents the number of lanes, and the 8 represents the lane size of the state block. As mentioned above, the input message from the ICAP interface should be 72 bits; therefore, the bit rate of this SHA3 implementation is set to 72 bits, and the reset 128 bits should be the capacity (defined by SHA3 reference). The round count for this SHA3 implementation, according to the provided formula, should be 12+2*3 = 18 rounds.

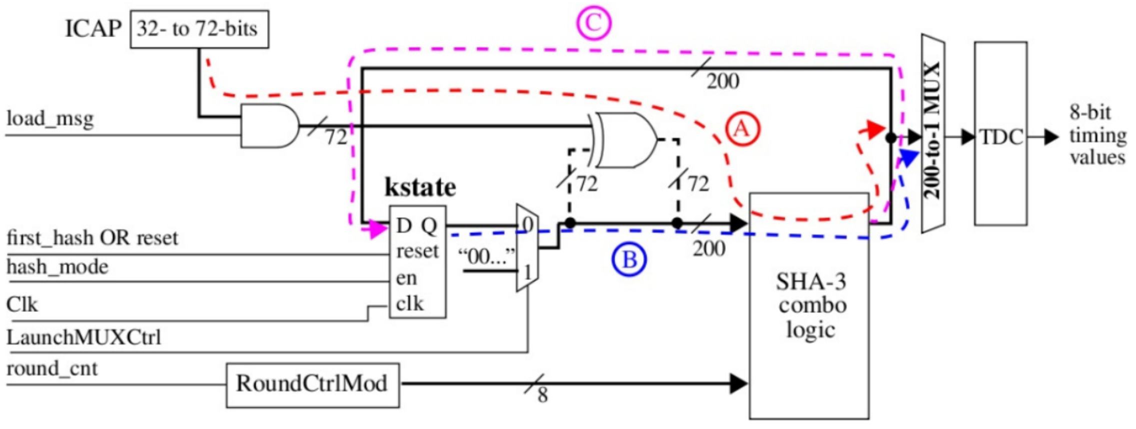


Figure 5 - Customized Controller Design and Path Timing[1]

Furthermore, a customized controller needs to be implemented to switch the SHA3 function between the Function and the PUF Mode. As shown above in Figure 5, two paths will be timed, which are shown as 'A' and 'B'. For the timing of path 'A', LaunchMUXCtrl and load_msg need to be asserted, and the hash_mode need to be de-asserted, so that the data from ICAP can be used separately for the timing purpose without being involved into the actual SHA3 operation. For the timing of 'B', the digest from the SHA3 is used to determine the timing information. The load_msg and hash_mode need to be de-asserted, and the LaunchMUXCtrl needs to be asserted, so that the saved digest from the SHA3 can be inserted to the SHA3 combo logic for path timing.

For the normal operation of SHA3 (Function Mode), the data will follow path 'C', and the load_msg will be asserted.

TDC

1. Carry Chain

The Carray Chain part is used to time the propagation delay. It consists of 32 CARRY4 components, which in total have 128 delay stages, from CC0 to CC127. The output of the SHA3 will propagate starting from CC0, and the propagation status will be stored into the Flip-Flops when the MPSClk signal arrives. Ideally, the MPSClk signal should arrive at a proper time to lock the buffer status into Flip-Flops, without having all 0s or 1s (underflow or overflow) in the locked status. The number of 0s in the locked status will be saved into the BRAM and used as the delay data.

When the timing process starts, the launch signal will be set up simultaneously with the start of path ‘A’ or ‘B’. With a properly selected tap point of MPS, the time delay can be measured by the carry chain without underflow or overflow. When the MPSClk signal arrives, the state in the carry chain will be stored, with upper Flip-Flops having 0s, and lower Flip-Flops having 1s. Lastly, the stored status will be decoded for getting TVal. The formula for the final timing value is PN = TVal + MPS Offset.

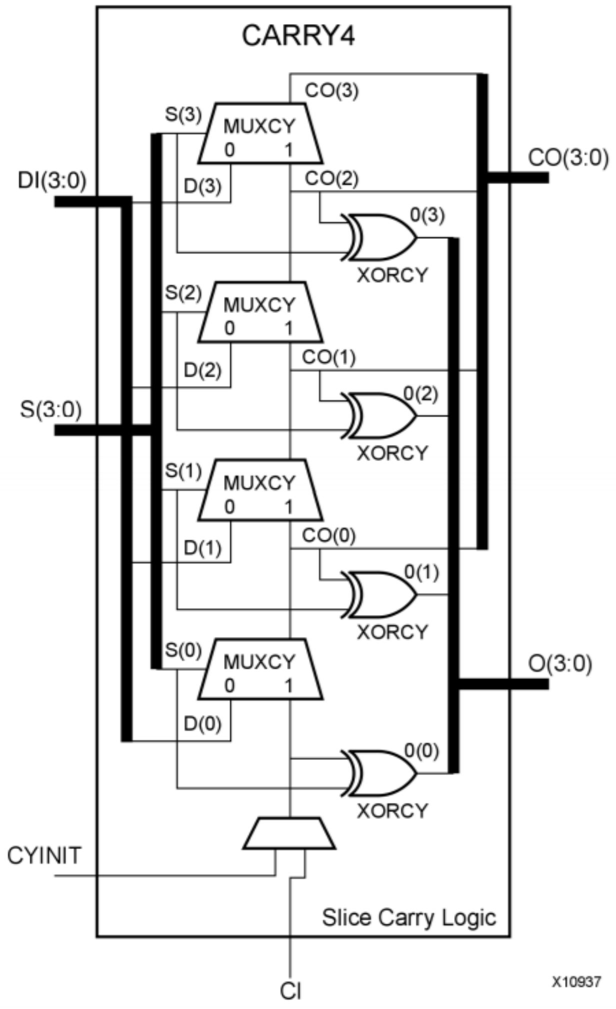


Figure 6 - CARRY4 Primitive[3]

For the very first CARRY4 block, the path signal is transit through CYINIT, while for all other CARRY4 blocks, the signal propagates from CO to CI. When propagating through these carry chains, output signals CO will be stored by D flip-flops (ThermFF). The selection bit S of every multiplexer is set to logic 1, in order to let the input signal CI or CYINIT propagate through the line.

2. Major Phase Shift

The tap points of the MPS will be determined during calibration. The requirement for the tap points setup is that any path (1 to 32 LUT) can be timed by at least two consecutive tap points without underflow for overflow. The tap points setup would need many experiments and trials, which is straightforward but time consuming.

The Major Phase Shift (MPS) is implemented to avoid underflow and overflow of Carry Chain. It consists of 50 concatenated buffers, and 12 tap points are carefully chosen among them (15 points are selected for the actual implementation). The clock signal to lock the Carry Chain status will propagate along the buffers and then be directed out from the desired tap point, so that the arriving time of the clock signal can be controlled.

The tap points of the MPS will be determined with the calibration using the Test Path. The requirement for the tap points setup is that any path can be timed by at least two consecutive tap points without underflow for overflow. The tap points setup would need many experiments and trials, which is straightforward but time consuming. The calibration data collected is on Github, which will be further explained in the Documentation part of the report.

3. Test Paths

The test paths are used for the Major Phase Shift Calibration. There will be cases that underflow and overflow may happen, and calibration can prevent these cases. For the test paths, there will be 32 “switches”, each with 2 paralleled 2-to-1 MUX (similar to arbiter PUF). The SWcon signal (32 bits) will be used to set up the 32 switches with different configurations, which can be configured either to switch the signals between the parallel pair or to direct pass the signal to the next pair of “signal switch”.. For the calibration process, the Calsel will be asserted, TPsel will be used to select test paths in the order of increasing delays.

Ring Oscillator

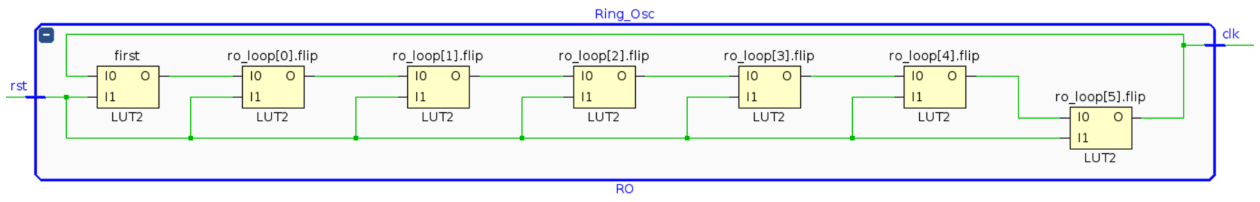


Figure 7 - Ring Oscillator Implementation

In order to avoid using clocks on the board, a ring oscillator is needed for generating clock signals. The implementation of the ring oscillator is shown as above in figure . The concatenated LUT2s will generate a raw clock signal of 120MHz. A clock generator can be used to generate clock signals with desired frequencies based on the raw 120 MHz ring oscillator clock signal.

BRAM

The timing data generated are stored in an on-board BRAM. For this implementation, BRAM_SINGLE_MACRO primitive is used for the BRAM storage. The storage size is chosen to be 36KB, and the read and write width are both 16 bits, which corresponds to 8 bit Carry Chain data + 8 bit MPS data. The detailed setup can be found in Xilinx Design Library Guide [2].

ICAP

I have tried to access the ICAP configuration port for two weeks, but still did not manage to get it to work. The Xilinx's documentation that I have followed is in the appendix. According to Xilinx, the configuration of ICAP is almost the same as SelectMAP, so maybe try to start with the SelectMAP section of the documentation. The part that we are most interested in is the "Readback and Configuration Verification". The integration for ICAP is listed as TODO in Top.vhd, as the ports are declared but commented out.

Others

For other parts that are not implemented, resources are provided in the appendix section for the theories and implementations of them.

HDL File Documentation For Timing Data Generation Integration

This section mainly focuses on the explanation of the HDL files for timing data generation integration. Please refer to the Github Repository while reading this section. The Github link can be found here.

SHA3_Absorb.vhd

This file is the Keccak-f[200] version of SHA3 combo logic. The implementation of it referred to the official Keccak team implementation, which can be found here.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
message_in	in	200	Message fed into the SHA3.
round_constant	in	8	Round constant for SHA3 operation, based on the current number of rounds of the SHA3 operation.
message_out	out	200	Hash result from SHA3.

SHA3_Function.vhd

This file builds connections of the SHA3 combo logic with TDC and ICAP interface. The operation is controlled by a finite state machine, which switches the SHA3 between operation mode and PUF mode. Under operation mode, it will operate as the normal SHA3 function, hashing the input message for 18 times with different round constant. Under PUF mode, the current SHA3 states will be locked into Flip-Flops, and it will be determined whether to time the delay from ICAP or just from SHA3 (Path ‘A’ or ‘B’). After the timing operation is done, if there is an overflow or underflow occurs in the Carry Chain, the entire timing operation will be redone using the same input to the SHA3 combo logic. Each of the 200 bits output from SHA3 will be timed.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
clk	in	1	Clock signal
data_ready	in	1	Flag to indicate if input data is ready for SHA3
from_ICAP	in	32	32 bits configuration data from ICAP
end_of_all_data	in	1	Flag to indicate if there is not more data left
data_request	out	1	Flag to indicate if SHA3 requests new data
to_CC	out	1	Signal output to Carry Chain, to measure propagation delay
Launch	out	1	Launch signal to MPS, to lock buffer status of Carry Chain
CC_finished	in	1	Flag to indicate if Carry Chain finishes the timing operation
CC_retry	in	1	Flag to indicate if a new tap point needs to be chosen due to overflow or underflow

PUF_start	in	1	Flag to indicate if the entire timing operation can start. (Test Path test is needed, and the timing operation needs to wait for Test Path test to finish)
timing_data_generation_finished	out	1	Flag to indicate if the entire timing data generation finishes

Carry_chain.vhd

This file constructs the Carry Chain with 32 cascaded CARRY4 primitives and 128 D Flip-Flops. Each of the CARRY4 primitive has 4 stages, which makes it in total of 4*32=128 stages. All the stages are connected with Flip-Flops, which would lock the status when the MPS clock signal arrives. After locking the signal, a process will be triggered to count the number of 0s, which will be directed out as TVal and used as the timing data from the Carry Chain.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
mpsclk	in	1	Signal from major phase shift to lock buffer status into the Flip-Flops
path	in	1	Signal input to propagate along buffers
finished	out	1	Flag to indicate if Carry Chain finishes timing operation
retry	out	1	Flag to indicate if overflow or underflow occurs
TVal	out	integer	Number of 0s in the locked Carry Chain status

Test_Path.vhd

This file is created to calibrate the MPS, and the main goal is to make sure that each path can be timed by 2 consecutive tap points. The test path is constructed by connecting 32 pairs of parallel 2-to-1 multiplexers, and these multiplexers are configured with the SWcon signal.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
SWcon	in	32	Use to configure test path configuration
Launch	in	1	Signal that would propagate through test path, to get the test path delay
TPsel	in	integer	Use to choose from which pair of “signal switch” the “Launch” signal would be directed out to the Carry Chain
test_path_o	out	1	“Launch” signal output from test path

MPS_Offset.vhd

The MPS is implemented by cascading 50 buffers together, so that the time delay of the output signal from MPS can be controlled. There are 15 tap points, which are selected according to the calibration results.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
MPSsel	in	8	Signal for selecting from which tap point should the clock signal be directed out to Carry Chain
Launch	in	1	Signal that would propagate through buffers, and be directed out to lock Carry Chain status
MPSClk	out	1	Output clock signal from MPS

TimeData.vhd

This file is an integration of the time-to-digital converter (TDC). The most important part is the handshaking protocol between SHA3 Function and TDC.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
clk	in	1	Clock signal
Launch	in	1	Launch signal to start the timing operation
to_CC	in	1	Signal from SHA3, which will be timed
testpath_passed	out	1	Flag to indicate if the test of test path is passed
finished	out	1	Flag to indicate if timing operation is finished
retry	out	1	Flag to indicate if overflow or underflow occurred
Time_Measured	out	16	Timing data measured

BRAM_test.vhd

This file is simply a mask for BRAM_SINGLE_MACRO primitive, to make it more friendly for the final integration. (Nothing special..)

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
DO	out	16	Data output
ADDR	in	12	Address
CLK	in	1	Clock signal

DI	in	16	Data input
EN	in	1	Enable signal of BRAM
REGCE	in	1	Output register clock enable input (won't be used)
RST	in	1	Reset signal for BRAM
WE	in	2	Byte write enable

RO.vhd

This file implemented a ring oscillator by concatenating 7 inverters together. The inverter is implemented using LUT1s. The generated clock signal has a frequency around 120 MHz.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
clk	out	1	Clock signal output from Ring Oscillator
rst	in	1	Reset signal

Key_Generation.vhd

This file is used to do the key generation. SHA3 is also used for the key generation. Timing data is read out from BRAM and then hashed together generating a 200-bit output. After the absorbing phase of SHA3, four squeeze operations are used to generate keys. Each squeeze output would have 72 usable bits for the key, and the final key is constructed as concatenating 4 squeeze outputs from SHA3.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
key_generation	in	1	Flag to indicate if key generation process can

_start			be started
clk	in	1	Clock signal
data_in	in	32	Data input for key generation
data_request	out	1	Flag to request more data for key generation
data_ready	in	1	Flag to indicate if the data requested is ready
no_more_data	in	1	Flag to indicate if no more data left for key generation
key_generated	out	256	Generated key
key_generation_finished	out	1	Flag to indicate if the key generation process is finished

Top.vhd

This file is the integration of TDC, SHA3 Function, BRAM, Ring Oscillator, and Key Generation.

Port Description			
Name	**Direction**	**Width (Bits)**	**Function**
key	out	256	The generated key
key_status	out	1	Flag to indicate if the currently key output is valid

Future Work

1. ICAP interface
2. Helper Data Generation
3. Integration and Test for Key Generation
4. Partial Configuration (SSBL)

Appendix

Xilinx Library Guide

7 Series FPGAs Configuration User Guide

ICAP Example

Helper Data Generation Paper

Reference

[1] D. Owen Jr. et al., "An Autonomous, Self-Authenticating, and Self-Contained Secure Boot Process for Field-Programmable Gate Arrays", *Cryptography*, vol. 2, no. 3, p. 15, 2018. Available: 10.3390/cryptography2030015 [Accessed 20 August 2021].

[2] G. Bertoni, J. Daemen, M. Peeters and G. Assche, "Keccak-reference-3.0", *Keccak*, 2021. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf. [Accessed: 20- Aug- 2021].

[3] "Vivado Design Suite 7 Series FPGA Libraries Guide", *Xilinx*, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf. [Accessed: 20- Aug- 2021].