FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, Lane, Shabal and Spectral Hash


Claude Shannon Institute for Discrete Mathematics, Coding and Cryptography & Department of Electrical & Electronic Engineering, University College Cork, Ireland.
Overview

- Hash Function Description
  - Introduction
  - Background
  - Operation
Overview

- Hash Function Description
  - Introduction
  - Background
  - Operation
- The SHA-3 Contest
Overview

- Hash Function Description
  - Introduction
  - Background
  - Operation
- The SHA-3 Contest
- Overview of the Hash Function Architectures
Overview

- Hash Function Description
  - Introduction
  - Background
  - Operation

- The SHA-3 Contest

- Overview of the Hash Function Architectures

- Hash Function Implementations
  - CubeHash
  - Grøstl
  - Lane
  - Shabal
  - Spectral Hash
Overview

- Hash Function Description
  - Introduction
  - Background
  - Operation
- The SHA-3 Contest
- Overview of the Hash Function Architectures
- Hash Function Implementations
  - CubeHash
  - Grøstl
  - Lane
  - Shabal
  - Spectral Hash
- Results and Conclusions
Modern communications networks such as WANs, LANs & the Internet handle vast amounts of sensitive information
Modern communications networks such as WANs, LANs & the Internet handle vast amounts of sensitive information.

The use of cryptographic algorithms is of crucial importance, to protect the privacy & integrity of this data.
Modern communications networks such as WANs, LANs & the Internet handle vast amounts of sensitive information.

The use of cryptographic algorithms is of crucial importance, to protect the privacy & integrity of this data.

Hash functions operate at the heart of contemporary cryptographic protocols:
- Digital Signature Standard (DSS)
- Transport Layer Security (TLS)
- Internet Protocol Security (IPSec)
- Random number generation algorithms
- Authentication algorithms
- Password storage mechanisms
Definition

A hash function $\mathcal{H}$ maps a message $x$ of variable length to a string of fixed length.
Operation of Hash Functions

**Definition**

A hash function \( H \) maps a message \( x \) of variable length to a string of fixed length. The process of applying \( H \) to \( x \) is called ‘hashing’.
Operation of Hash Functions

Definition

A hash function $\mathcal{H}$ maps a message $x$ of variable length to a string of fixed length. The process of applying $\mathcal{H}$ to $x$ is called ‘hashing’. The output $\mathcal{H}(x)$ is called the ‘message hash’ or ‘message digest’.
Properties of Hash Functions

Pre-image Resistance (one-wayness)

This requirement means that for a given hash value $y$, it should be computationally infeasible for an adversary to find an input $x$ such that $\mathcal{H}(x) = y$. 
Properties of Hash Functions

Pre-image Resistance (one-wayness)
This requirement means that for a given hash value $y$, it should be computationally infeasible for an adversary to find an input $x$ such that $H(x) = y$

Second Pre-image Resistance (weak collision resistance)
This implies that for a given input $x_1$, it should be computationally infeasible to find another input $x_2$, such that $H(x_1) = H(x_2)$
Properties of Hash Functions

Pre-image Resistance (one-wayness)
This requirement means that for a given hash value \( y \), it should be computationally infeasible for an adversary to find an input \( x \) such that \( H(x) = y \)

Second Pre-image Resistance (weak collision resistance)
This implies that for a given input \( x_1 \), it should be computationally infeasible to find another input \( x_2 \), such that \( H(x_1) = H(x_2) \)

(Strong) Collision Resistance
It should be computationally infeasible for an adversary to find any two distinct inputs \( x_1 \) and \( x_2 \), such that \( H(x_1) = H(x_2) \)
SHA-2

- Family of Secure Hash Algorithms (SHA) began in 1993
Family of Secure Hash Algorithms (SHA) began in 1993
The National Institute of Standards & Technology (NIST) published the Secure Hash Standard - SHA-0
SHA-2

- Family of Secure Hash Algorithms (SHA) began in 1993
- The National Institute of Standards & Technology (NIST) published the Secure Hash Standard - SHA-0
- Withdrawn by the National Security Agency (NSA) & replaced in 1995 by the SHA-1 algorithm
  - Both algorithms produce message hashes of 160-bits
Family of Secure Hash Algorithms (SHA) began in 1993

The National Institute of Standards & Technology (NIST) published the Secure Hash Standard - SHA-0

Withdrawn by the National Security Agency (NSA) & replaced in 1995 by the SHA-1 algorithm
  - Both algorithms produce message hashes of 160-bits

In 2002, NIST published three new hash functions with longer hash lengths:
  - SHA-256
  - SHA-384
  - SHA-512
SHA-2

- Family of Secure Hash Algorithms (SHA) began in 1993
- The National Institute of Standards & Technology (NIST) published the Secure Hash Standard - SHA-0
- Withdrawn by the National Security Agency (NSA) & replaced in 1995 by the SHA-1 algorithm
  - Both algorithms produce message hashes of 160-bits
- In 2002, NIST published three new hash functions with longer hash lengths:
  - SHA-256
  - SHA-384
  - SHA-512
  - In 2004, SHA-224 was added to the standard
Family of Secure Hash Algorithms (SHA) began in 1993
- The National Institute of Standards & Technology (NIST) published the Secure Hash Standard - SHA-0
- Withdrawn by the National Security Agency (NSA) & replaced in 1995 by the SHA-1 algorithm
  - Both algorithms produce message hashes of 160-bits
- In 2002, NIST published three new hash functions with longer hash lengths:
  - SHA-256
  - SHA-384
  - SHA-512
  - In 2004, SHA-224 was added to the standard
- These four algorithms form the SHA-2 family of hash functions
The SHA-3 Contest

Currently, NIST is holding a public competition to develop a new cryptographic hash algorithm(s)
The SHA-3 Contest

- Currently, NIST is holding a public competition to develop a new cryptographic hash algorithm(s)
- The new hash function(s) will be called SHA-3 (or the Advanced Hash Standard (AHS))
  - Which may ultimately supersede the functions in the SHA-2 family
Currently, NIST is holding a public competition to develop a new cryptographic hash algorithm(s)

The new hash function(s) will be called SHA-3 (or the Advanced Hash Standard (AHS))
  - Which may ultimately supersede the functions in the SHA-2 family

The SHA-3 Contest - Current State

The contest has received 64 submissions from designers all around the world.
The SHA-3 Contest- Current State

- The contest has received 64 submissions from designers all around the world
- 51 of these designs recently progressed through to the first round of the contest
The SHA-3 Contest- Current State

- The contest has received 64 submissions from designers all around the world
- 51 of these designs recently progressed through to the first round of the contest
- So far, 10 out of 51 first round candidates have been officially conceded broken or withdrawn by the designers
The SHA-3 Contest- Current State

- The contest has received 64 submissions from designers all around the world.
- 51 of these designs recently progressed through to the first round of the contest.
- So far, 10 out of 51 first round candidates have been officially conceded broken or withdrawn by the designers.
- Over the coming years the number of candidate designs will be reduced by NIST.
The SHA-3 Contest - Current State

- The contest has received 64 submissions from designers all around the world.
- 51 of these designs recently progressed through to the first round of the contest.
- So far, 10 out of 51 first round candidates have been officially conceded broken or withdrawn by the designers.
- Over the coming years the number of candidate designs will be reduced by NIST.
- It is planned to announce the successful hash function(s) in 2012.
The SHA-3 Contest

- These hash algorithms are available for public comment and scrutiny
  - (such research is vital to the selection process)
The SHA-3 Contest

- These hash algorithms are available for public comment and scrutiny
  (such research is vital to the selection process)
- NIST has stated that computational efficiency of the algorithms in hardware, over a wide range of platforms, will be addressed during the second round of the contest
FPGA’s are an attractive choice for implementing cryptographic algorithms.
Field Programmable Gate Array’s

- FPGAs are an attractive choice for implementing cryptographic algorithms
  - Low cost relative to ASICs
  - Flexible when adopting security protocol upgrades as they can be re-programmed in-place
  - Allow rapid prototyping of designs
Field Programmable Gate Array’s

- FPGAs are an attractive choice for implementing cryptographic algorithms
  - Low cost relative to ASICs
  - Flexible when adopting security protocol upgrades as they can be re-programmed in-place
  - Allow rapid prototyping of designs
- Goal is to explore area-speed trade-offs in the five implementations
FPGAs are an attractive choice for implementing cryptographic algorithms
- Low cost relative to ASICs
- Flexible when adopting security protocol upgrades as they can be re-programmed in-place
- Allow rapid prototyping of designs

Goal is to explore area-speed trade-offs in the five implementations

Compare the hardware efficiency of the designs by examining the throughput per unit area metric
FPGA Comparison

Two FPGA platforms were targeted in the study:

- The low-cost Xilinx Spartan-3 \texttt{xc3s5000-5fg900}
- The newer technology Xilinx Virtex-5 \texttt{xc5vlx220-2-ff1760}
FPGA Comparison

Two FPGA platforms were targeted in the study:

- The low-cost Xilinx Spartan-3 \( xc3s5000-5fg900 \)
- The newer technology Xilinx Virtex-5 \( xc5vlx220-2-ff1760 \)

Results for a particular hash function architecture on the two FPGA platforms cannot be directly compared due to the different underlying technologies.
FPGA Comparison

- Two FPGA platforms were targeted in the study:
  - The low-cost Xilinx Spartan-3 xc3s5000-5fg900
  - The newer technology Xilinx Virtex-5 xc5vlx220-2-ff1760
- Results for a particular hash function architecture on the two FPGA platforms cannot be directly compared due to the different underlying technologies
- Post-Place and Route results
Design Goals

- Our main design goal was to optimise throughput
Design Goals

- Our main design goal was to optimise throughput

Throughput

\[
\text{Throughput} = \frac{\text{# Bits in a message block} \times \text{Maximum clock frequency}}{\text{# Clock cycles per message block}}
\]
Design Goals

- Our main design goal was to optimise throughput

Throughput

\[
\text{Throughput} = \frac{\text{# Bits in a message block} \times \text{Maximum clock frequency}}{\text{# Clock cycles per message block}}
\]

- High-throughput hash function implementations are beneficial in network server applications
Our main design goal was to optimise throughput

Throughput = \frac{\# \text{ Bits in a message block} \times \text{Maximum clock frequency}}{\# \text{ Clock cycles per message block}}

- High-throughput hash function implementations are beneficial in network server applications
- A secondary goal was to analyse the throughput per slice of the architectures
  - Determines which hash function implementations make the most efficient use of FPGA area
Core Functionality

- All of the hash algorithms examined include initialisation & padding, compression & message output stages
Core Functionality

- All of the hash algorithms examined include initialisation & padding, compression & message output stages.
- We focused on implementing the compression function $f$ of each hash function.
  - The compression functions perform the majority of the computations in the hash algorithms.
  - The throughput of the algorithms is largely determined by the throughput of the compression functions.
CubeHash

CubeHash
CubeHash

- CubeHash was submitted by Dan Bernstein
- University of Illinois at Chicago, Department of Computer Science
- http://cubehash.cr.yp.to/
CubeHash Overview

- CubeHash is defined by three parameters:
  - $b \in \{1, 2, 3, \ldots, 128\}$, the number of bytes in a block of the padded message
  - $r \in \{1, 2, 3, \ldots\}$, the number of times the compression function is iterated for each padded message block
  - $h \in \{8, 16, 24, \ldots, 512\}$, the number of bits in the message digest
CubeHash Overview

- CubeHash is defined by three parameters:
  - $b \in \{1, 2, 3, \ldots, 128\}$, the number of bytes in a block of the padded message
  - $r \in \{1, 2, 3, \ldots\}$, the number of times the compression function is iterated for each padded message block
  - $h \in \{8, 16, 24, \ldots, 512\}$, the number of bits in the message digest

- All of the CubeHash variants use a 1024-bit state represented as thirty-two 32-bit words $x[t], 0 \leq t \leq 31$
CubeHash Overview

- CubeHash is defined by three parameters:
  - \( b \in \{1, 2, 3, \ldots, 128\} \), the number of bytes in a block of the padded message
  - \( r \in \{1, 2, 3, \ldots\} \), the number of times the compression function is iterated for each padded message block
  - \( h \in \{8, 16, 24, \ldots, 512\} \), the number of bits in the message digest
- All of the CubeHash variants use a 1024-bit state represented as thirty-two 32-bit words \( x[t], 0 \leq t \leq 31 \)
- During the algorithm’s execution, the state is operated upon by a compression function, denoted here by \( f_C \)
CubeHash Design

The compression function has two 512-bit inputs, $A$ & $B$.
2 × 16 additions modulo $2^{32}$ (denoted by $\boxplus$), where the datapath $A$ is added word-by-word to datapath $B$
$2 \times 16$ 32-bit Boolean XORs (denoted by $\oplus$), where the two datapaths are XORed word-by-word
$2 \times 16$ rotation operations, where each word in datapath $B$ is cyclically rotated upwards by a fixed number of bits.
**CubeHash Design**

$4 \times 8$ swapping operations, where specified words in a datapath exchange positions.
CubeHash Results

- $f_C$ is used $r = 8$ times for each message block $M_i$
- i.e. for each message byte in this case, since $b = 1$
CubeHash Results

- $f_C$ is used $r = 8$ times for each message block $M_i$
  - i.e. for each message byte in this case, since $b = 1$
- We investigated CubeHash architectures where $f_C$ is unrolled by various degrees
CubeHash Results

- $f_C$ is used $r = 8$ times for each message block $M_i$
  - i.e. for each message byte in this case, since $b = 1$
- We investigated CubeHash architectures where $f_C$ is unrolled by various degrees
  - The lowest area design iteratively uses a single $f_C$ unit & takes 8 clock cycles to process a single message block
  - The highest area design uses a chain of four $f_C$ units in series to process a single message block in two clock cycles
CubeHash Results

- $f_C$ is used $r = 8$ times for each message block $M_i$
  - i.e. for each message byte in this case, since $b = 1$
- We investigated CubeHash architectures where $f_C$ is unrolled by various degrees
  - The lowest area design iteratively uses a single $f_C$ unit & takes 8 clock cycles to process a single message block
  - The highest area design uses a chain of four $f_C$ units in series to process a single message block in two clock cycles
- It is interesting to note that the throughput on both FPGAs is quite low, because each message block in CubeHash consists of only 1 byte
Grøstl
Grøstl

- Grøstl was designed by a team of cryptographers from Technical University of Denmark (DTU) and TU Graz
- Praveen Gauravaram and Lars R. Knudsen and Krystian Matusiewicz and Florian Mendel and Christian Rechberger and Martin Schläffer and Søren S. Thomsen
- [http://www.groestl.info](http://www.groestl.info)
Grøstl Overview

It is an iterated hash function with a compression function built from two fixed, large distinct permutations.
Grøstl Overview

- It is an iterated hash function with a compression function built from two fixed, large distinct permutations.
- It borrows heavily from components used in the AES block cipher.
Grøstl Overview

- It is an iterated hash function with a compression function built from two fixed, large distinct permutations.
- It borrows heavily from components used in the AES block cipher.
- Four variants of Grøstl are explicitly defined for the SHA-3 contest, where $n \in \{224, 256, 384, 512\}$.
Grøstl Overview

- It is an iterated hash function with a compression function built from two fixed, large distinct permutations.
- It borrows heavily from components used in the AES block cipher.
- Four variants of Grøstl are explicitly defined for the SHA-3 contest, where $n \in \{224, 256, 384, 512\}$.
- The Grøstl variants can be divided into two categories, based on the size of the algorithm’s internal state, denoted by $\ell$. 
Grøstl Overview

- It is an iterated hash function with a compression function built from two fixed, large distinct permutations.
- It borrows heavily from components used in the AES block cipher.
- Four variants of Grøstl are explicitly defined for the SHA-3 contest, where \( n \in \{224, 256, 384, 512\} \).
- The Grøstl variants can be divided into two categories, based on the size of the algorithm’s internal state, denoted by \( \ell \):
  - Grøstl-224/256 \( \ell = 512 \)
  - Grøstl-384/512 \( \ell = 1024 \).
Grøstl Overview

- It is an iterated hash function with a compression function built from two fixed, large distinct permutations
- It borrows heavily from components used in the AES block cipher
- Four variants of Grøstl are explicitly defined for the SHA-3 contest, where $n \in \{224, 256, 384, 512\}$
- The Grøstl variants can be divided into two categories, based on the size of the algorithm’s internal state, denoted by $\ell$
  - Grøstl-224/256 $\ell = 512$
  - Grøstl-384/512 $\ell = 1024$

we denote the Grøstl compression function by $f_G$
Grøstl Design

Each message block $M_i$ is combined with the previous hash value $H_{i-1}$.
A permutation $P$ is applied to the result

$$\begin{array}{c}
1 \xrightarrow{\text{sel}} 0 \\
\Downarrow \text{AddRoundConstant} \\
\Downarrow \text{SubBytes} \\
\Downarrow \text{ShiftBytes} \\
\Downarrow \text{MixBytes}
\end{array}$$
A second permutation $Q$ operates in parallel on $M_i$. 

\[ H_{i-1} \rightarrow P \rightarrow Q \rightarrow M_i \]

\[ f_G \rightarrow H_i \]
Grøstl Design

The compression function output $H_i$ is formed by XOR-ing $H_{i-1}$ with the outputs of $P$ & $Q$. 

![Diagram of Grøstl Design](image)
Grøstl Design

The function $P(x) \oplus x$ is applied, where $x$ is the final hash value $H_{N-1}$, & the $\ell$-bit result is truncated to leave the rightmost $n$ bits.
Grøstl Results

We implemented the S-boxes as look-up tables, & investigated two design choices:
Grøstl Results

We implemented the S-boxes as look-up tables, & investigated two design choices:

- Storage of the S-boxes in distributed memory (FPGA slice LUTs)
- Storage in dedicated Block RAM (BRAM)
Grøstl Results

We implemented the S-boxes as look-up tables, & investigated two design choices:
- Storage of the S-boxes in distributed memory (FPGA slice LUTs)
- Storage in dedicated Block RAM (BRAM)

Implementing using BRAM, available on both FPGAs, improves the clock frequency significantly while also reducing the number of slices required

This returns a higher throughput/area metric for both Grøstl variants
We implemented the S-boxes as look-up tables, & investigated two design choices:

- Storage of the S-boxes in distributed memory (FPGA slice LUTs)
- Storage in dedicated Block RAM (BRAM)

Implementing using BRAM, available on both FPGAs, improves the clock frequency significantly while also reducing the number of slices required.

This returns a higher throughput/area metric for both Grøstl variants.

For Grøstl-384/512, the BRAM requirements of the parallel architectures exceeds the resources available on the FPGAs.
Lane

Lane
Lane

Lane was submitted by the COSIC research group of the Katholieke Universiteit Leuven, Belgium

Sebastiaan Indesteege, Elena Andreeva, Christophe De Cannière, Orr Dunkelman, Emilia Käsper, Svetla Nikova, Bart Preneel, Elmar Tischhauser

http://www.cosic.esat.kuleuven.be/lane/
Components of the AES block cipher are also used in the Lane compression function.
Lane Overview

- Components of the AES block cipher are also used in the Lane compression function.
- The Lane variants can be divided into two categories, according to the size of the internal states:
  - Lane-224 & Lane-256 use a 256-bit state (corresponding to 2 AES states)
  - Lane-384 & Lane-512 use a 512-bit state (corresponding to 4 AES states)
Components of the AES block cipher are also used in the Lane compression function.

The Lane variants can be divided into two categories, according to the size of the internal states:

- Lane-224 & Lane-256 use a 256-bit state (corresponding to 2 AES states)
- Lane-384 & Lane-512 use a 512-bit state (corresponding to 4 AES states)

We denote the Lane compression function by $f_L$. 
$f_L$ begins with a message expansion, where $M_i$ is combined with $H_{i-1}$ using a series of XOR & concatenation operations.
The result of the message expansion stage is six 256-bit expanded message words $W_0, \ldots, W_5$. 

![Diagram of the message expansion process with $W_0$ to $W_5$ feeding into the Lane design process.]
Lane Design

The remainder of $f_C$ consists of five 256-bit XOR operations & eight permutation ‘lanes’, $P_i$, $i \in \{0, \ldots, 5\}$, & $Q_j$, $j \in \{0, 1\}$, arranged in two layers.
Each permutation is executed $r$ times. e.g. $r = 6$ for $P_i$ & $r = 3$ for $Q_i$ for Lane-224/256
Permutation blocks $P_i$ & $Q_j$ are identical $\forall (i,j)$
The SubBytes, ShiftRows & MixColumns transformations are re-used from the AES block cipher, & are applied independently to each AES sub-state within the $P_i$ or $Q_j$ state
The AddConstant transformation XOR's a pre-defined round constant $k_i$ with part of the $P_i/Q_j$ state
The AddCounter transformation XORs part of the counter $C_i$ with the state.
The SwapColumns transformation swaps columns between the AES sub-states that make up the $P_i/Q_j$ state.
Lane Results

- A high-area implementation of $f_L$ uses six permutation circuits, to process each $W_i$ in parallel
  - Two of these circuits can be re-used to calculate $Q_1$ & $Q_2$ without loss of speed
Lane Results

- A high-area implementation of $f_L$ uses six permutation circuits, to process each $W_i$ in parallel
  - Two of these circuits can be re-used to calculate $Q_1$ & $Q_2$ without loss of speed
- A low-area implementation of $f_L$ uses just one permutation circuit to compute $P_i$ & $Q_j, \forall (i, j)$
  - This requires extra control circuitry & extra clock cycles
Lane Results

- A high-area implementation of $f_L$ uses six permutation circuits, to process each $W_i$ in parallel.
  - Two of these circuits can be re-used to calculate $Q_1$ & $Q_2$ without loss of speed.
- A low-area implementation of $f_L$ uses just one permutation circuit to compute $P_i$ & $Q_j \forall (i, j)$.
  - This requires extra control circuitry & extra clock cycles.

We investigated this trade-off by implementing with various numbers of parallel permutation lanes.
Lane Results

- A high-area implementation of $f_L$ uses six permutation circuits, to process each $W_i$ in parallel.
  - Two of these circuits can be re-used to calculate $Q_1$ & $Q_2$ without loss of speed.
- A low-area implementation of $f_L$ uses just one permutation circuit to compute $P_i$ & $Q_j \forall (i, j)$.
  - This requires extra control circuitry & extra clock cycles.
- We investigated this trade-off by implementing with various numbers of parallel permutation lanes.
- The more $P_i$ blocks that are implemented in parallel, the better the throughput attained.
Lane Results

- A high-area implementation of $f_L$ uses six permutation circuits, to process each $W_i$ in parallel
  - Two of these circuits can be re-used to calculate $Q_1$ & $Q_2$ without loss of speed
- A low-area implementation of $f_L$ uses just one permutation circuit to compute $P_i$ & $Q_j \forall (i, j)$
  - This requires extra control circuitry & extra clock cycles
- We investigated this trade-off by implementing with various numbers of parallel permutation lanes
- The more $P_i$ blocks that are implemented in parallel, the better the throughput attained
- The LANE-384/512 6 $P_i$ blocks parallel architectures exceeds the resources available on the Spartan 3
Shabal

Shabal
Shabal

- Shabal was submitted by the Saphir research project, France
- Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, María Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, Marion Videau
- http://www.shabal.com/
Shabal Overview

- It uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits
Shabal Overview

- It uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits.
- The hash function output is of size $\ell_h$ bits, where $\ell_h \in \{192, 224, 256, 384, 512\}$.
Shabal Overview

- It uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits.
- The hash function output is of size $\ell_h$ bits, where $\ell_h \in \{192, 224, 256, 384, 512\}$.
- The compression function $f_{Sh}$, is based on a Non-Linear Feedback Shift Register (NLFSR) construction.
Shabal Overview

- It uses a sequential iterative hash construction, to process messages in blocks of $\ell_m = 512$ bits.
- The hash function output is of size $\ell_h$ bits, where $\ell_h \in \{192, 224, 256, 384, 512\}$.
- The compression function $f_{Sh}$, is based on a Non-Linear Feedback Shift Register (NLFSR) construction.
- This compression function operates on an internal state, denoted by $(A, B, C)$:
  - $A$ is defined as a $12 \times 32$-bit word.
  - $B$ & $C$ are defined as $16 \times 32$-bit words.
  - A 64-bit counter $W$ is defined to track the number of message blocks that have been processed.
Each message block $M_w$ is combined with state words $B$ & $C$ using addition & subtraction modulo $2^{32}$.
State word $A$ is XORed with the counter $W$
The permutation $\mathcal{P}$ has three sequential operations:

- $V$ followed by $A$ with a circular shift of 15 bits to the left.
- $U$ followed by $C$ with a circular shift of 8 bits to the left.
- $M$ with a circular shift of 6 bits to the left.
- $B$ with a circular shift of 9 bits to the left.
- $\ll 1$

Shabal Design
Shabal Design

Rotation: $B[i] \leftarrow B[i] \ll 17$ for $0 \leq i \leq 15$
Shabal Design

NLFSR-based Permutation:
Hash Functions
The SHA-3 Contest
Hash Function Implementations
Results
Conclusions

Shabal Design

\[
A[i + 16j \mod 12] \leftarrow U(A[i + 16j \mod 12] \oplus V(A[(i - 1) + 16j \mod 12] \ll 15) \oplus C[8 - i \mod 16]) \oplus B[i + 13 \mod 16] \oplus (B[i + 9 \mod 16] \land B[i + 6 \mod 16] \oplus M[i])
\]

\[
B[i] \leftarrow (B[i] \ll 1) \oplus A[i + 16j \mod 12]
\]
Shabal Design

Addition: $A[j \mod 12] \leftarrow A[j \mod 12] \boxplus C[j + 3 \mod 16]$ for $0 \leq j \leq 35$
Shabal Results

- The final part of $\mathcal{P}$ adds words from the $A$ & $C$ states
Shabal Results

- The final part of $\mathcal{P}$ adds words from the $A$ & $C$ states
- For these modulo $2^{32}$ additions, we investigated two designs
Shabal Results

- The final part of $P$ adds words from the $A$ & $C$ states
- For these modulo $2^{32}$ additions, we investigated two designs
- The NLFSR is used together with a single adder to compute $A[0] \oplus C[3]$, & the result is fed back to $A[15]$
  - This design takes a further 36 clock cycles to produce the final result
Shabal Results

- The final part of $P$ adds words from the $A$ & $C$ states
- For these modulo $2^{32}$ additions, we investigated two designs
- The NLFSR is used together with a single adder to compute $A[0] \oplus C[3]$, & the result is fed back to $A[15]$
  - This design takes a further 36 clock cycles to produce the final result
  - The final result is computed without requiring any extra clock cycles, but at the expense of area for 35 additional adders
Shabal Results

- The final part of $P$ adds words from the $A$ & $C$ states
- For these modulo $2^{32}$ additions, we investigated two designs
- The NLFSR is used together with a single adder to compute $A[0] \oplus C[3]$, & the result is fed back to $A[15]$
  - This design takes a further 36 clock cycles to produce the final result
  - The final result is computed without requiring any extra clock cycles, but at the expense of area for 35 additional adders
- The higher-area implementation is more efficient on the Spartan-3 platform, & the lower-area implementation is more efficient on the Virtex-5 platform
Spectral Hash
Spectral Hash

Spectral Hash ‘s-hash’ was submitted by the College of Creative Studies & Department of Computer Science University of California Santa Barbara

Gokay Saldamlı, Cevahir Demirkıran, Megan Maguire, Carl Minden, Jacob Topper, Alex Troesch, Cody Walker, Çetin Kaya Koç

Spectral Hash

It is a Merkle-Damgård based hash function which uses the Discrete Fourier Transform (DFT) to generate the required diffusion & confusion properties.
Spectral Hash

- It is a Merkle-Damgård based hash function which uses the Discrete Fourier Transform (DFT) to generate the required diffusion & confusion properties.

- The compression function $f_{Sp}$ operates on:
  - The current message block $M_i$
  - The hash of the previous message $H_{i-1}$
  - A permutation state $P_i$, which is dependent on previously processed message blocks
Spectral Hash

- It is a Merkle-Damgård based hash function which uses the Discrete Fourier Transform (DFT) to generate the required diffusion & confusion properties.

- The compression function $f_{Sp}$ operates on:
  - The current message block $M_i$
  - The hash of the previous message $H_{i-1}$
  - A permutation state $P_i$, which is dependent on previously processed message blocks

- These internal states are termed *prisms*
Spectral Hash

- It is a Merkle-Damgård based hash function which uses the Discrete Fourier Transform (DFT) to generate the required diffusion & confusion properties.

- The compression function $f_{Sp}$ operates on:
  - The current message block $M_i$
  - The hash of the previous message $H_{i-1}$
  - A permutation state $P_i$, which is dependent on previously processed message blocks.

- These internal states are termed *prisms*.

- These are represented as $4 \times 4 \times 8$ matrices of different word sizes.
Spectral Hash Design

The S-Hash compression function $f_{Sp}$ consists of the following operations on both the $s$-prism & the $p$-prism:

1. Affine Transformation
2. $K-DFT$
3. $J-DFT$
4. $I-DFT$
5. Nonlinear Transformation
6. Plane Rotation
7. Data Register
8. Permutation Register

Input:
- HashIn
- MessageIn

Output:
- MessageOut
- PermutationOut
Affine Transformation: performs an inversion in $GF(2^4)$ followed by a linear shift on each byte in $p$-prism
Swap Controls: swap bytes in $p$-prism according to the values in $s$-prism

![Diagram of Spectral Hash Design]
Discrete Fourier Transforms: sixteen 8-point DFTs & two sets of thirty-two 4-point DFTs are performed on $s$-prism
Non-linear Transformation: a transformation of \textit{s-prism} according to the state of \textit{p-prism}. The previous hash value \textit{h-prism} is also taken into account in this step.
Spectral Hash Design

Plane Rotation: an output rotation of the *p-prism*
Spectral Hash Results

- Two implementations, both using Look-Up-Tables (LUT) for the affine transformation & the modular 17 reduction in the DFT calculations
Spectral Hash Results

- Two implementations, both using Look-Up-Tables (LUT) for the affine transformation & the modular 17 reduction in the DFT calculations.
- The *Single Cycle* implementation completes $f_{Sp}$ in a single clock cycle by performing the *s-prism* & *p-prism* calculations in parallel & fully unrolling the operations.
  - This leads to a large area, i.e. sixteen 8-point DFT hardware architectures.
Spectral Hash Results

- Two implementations, both using Look-Up-Tables (LUT) for the affine transformation & the modular 17 reduction in the DFT calculations.

- The *Single Cycle* implementation completes $f_{Sp}$ in a single clock cycle by performing the $s$-prism & $p$-prism calculations in parallel & fully unrolling the operations.
  - This leads to a large area, i.e. sixteen 8-point DFT hardware architectures.

- The *Low Area* architecture reuses the hardware, saving on area at a cost of extra clock cycles & control circuitry.
Spectral Hash Results

- Two implementations, both using Look-Up-Tables (LUT) for the affine transformation & the modular 17 reduction in the DFT calculations.
- The *Single Cycle* implementation completes $f_{Sp}$ in a single clock cycle by performing the $s$-prism & $p$-prism calculations in parallel & fully unrolling the operations.
  - This leads to a large area, i.e. sixteen 8-point DFT hardware architectures.
- The *Low Area* architecture reuses the hardware, saving on area at a cost of extra clock cycles & control circuitry.
- On the Virtex-5, very high throughputs can be attained using a fully parallel implementation.
Spectral Hash Results

- Two implementations, both using Look-Up-Tables (LUT) for the affine transformation & the modular 17 reduction in the DFT calculations.
- The *Single Cycle* implementation completes $f_{Sp}$ in a single clock cycle by performing the *s-prism* & *p-prism* calculations in parallel & fully unrolling the operations.
  - This leads to a large area, i.e. sixteen 8-point DFT hardware architectures.
- The *Low Area* architecture reuses the hardware, saving on area at a cost of extra clock cycles & control circuitry.
- On the Virtex-5, very high throughputs can be attained using a fully parallel implementation.
- This parallel architectures exceeds the resources available on the Spartan 3.
Comparison of Implementations
In order to compare the various hash function implementations with each other, the throughput and area results for all of the designs presented were plotted, for the Spartan-3 and Virtex-5 FPGAs.
In order to compare the various hash function implementations with each other, the throughput and area results for all of the designs presented were plotted, for the Spartan-3 and Virtex-5 FPGAs.

In each figure, the results are labelled 1–21 as they were presented:

- The crosses (×) denote designs that produce only 224/256-bit hashes.
- The stars (∗) denote designs that produce only 384/512-bit hashes.
- The bullets (●) denote designs that can produce all of the required hash lengths.
**Spartan-3**

![Graph showing throughput and area for various hash function implementations]

<table>
<thead>
<tr>
<th>Design Type</th>
<th>TP/Area (Mbps/slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cubehash</strong></td>
<td></td>
</tr>
<tr>
<td>#1 Iterative</td>
<td>0.02</td>
</tr>
<tr>
<td>#2 2x-unrolled</td>
<td>0.02</td>
</tr>
<tr>
<td>#3 4x-unrolled</td>
<td>0.01</td>
</tr>
<tr>
<td><strong>Grøstl</strong> 224/256</td>
<td></td>
</tr>
<tr>
<td>#4 BRAM-Interleaved</td>
<td>0.73</td>
</tr>
<tr>
<td>#5 BRAM-Parallel</td>
<td>0.75</td>
</tr>
<tr>
<td>#6 Slice LUTs-Interleaved</td>
<td>0.28</td>
</tr>
<tr>
<td>#7 Slice LUTs-Parallel</td>
<td>0.30</td>
</tr>
<tr>
<td><strong>Grøstl</strong> 384/512</td>
<td></td>
</tr>
<tr>
<td>#8 BRAM-Interleaved</td>
<td>0.46</td>
</tr>
<tr>
<td>#9 BRAM-Parallel</td>
<td>-</td>
</tr>
<tr>
<td>#10 Slice LUTs-Interleaved</td>
<td>0.17</td>
</tr>
<tr>
<td>#11 Slice LUTs-Parallel</td>
<td>0.18</td>
</tr>
</tbody>
</table>
Spartan-3

<table>
<thead>
<tr>
<th>Design</th>
<th>Type</th>
<th>TP/Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane Parallel $P_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#12</td>
<td>1</td>
<td>0.09</td>
</tr>
<tr>
<td>#13</td>
<td>2</td>
<td>0.08</td>
</tr>
<tr>
<td>#14</td>
<td>6</td>
<td>0.05</td>
</tr>
<tr>
<td>Lane Parallel $P_i$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#15</td>
<td>1</td>
<td>0.10</td>
</tr>
<tr>
<td>#16</td>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td>#17</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Shabal Final Additions in $P$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#18</td>
<td>Series</td>
<td>0.27</td>
</tr>
<tr>
<td>#19</td>
<td>Parallel</td>
<td>0.33</td>
</tr>
<tr>
<td>S-Hash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#20</td>
<td>Low Area</td>
<td>0.009</td>
</tr>
<tr>
<td>#21</td>
<td>Single Cycle</td>
<td>-</td>
</tr>
</tbody>
</table>

The Claude Shannon Workshop On Coding and Cryptography
Virtex-5

<table>
<thead>
<tr>
<th>Design</th>
<th>Type</th>
<th>TP/Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>CubeHash</td>
<td>#1 Iterative</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>#2 2x-unrolled</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>#3 4x-unrolled</td>
<td>0.07</td>
</tr>
<tr>
<td>Grostl</td>
<td>224/256 BRAM-Interleaved</td>
<td>2.01</td>
</tr>
<tr>
<td></td>
<td>#5 BRAM-Parallel</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>#6 Slice LUTs-Interleaved</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>#7 Slice LUTs-Parallel</td>
<td>0.63</td>
</tr>
<tr>
<td>Grostl</td>
<td>384/512 BRAM-Interleaved</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>#9 BRAM-Parallel</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>#10 Slice LUTs-Interleaved</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>#11 Slice LUTs-Parallel</td>
<td>0.31</td>
</tr>
</tbody>
</table>
Virtex-5

<table>
<thead>
<tr>
<th>Design</th>
<th>Type</th>
<th>TP/Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lane Parallel $P_i$</td>
<td></td>
</tr>
<tr>
<td>#12</td>
<td>1</td>
<td>0.40</td>
</tr>
<tr>
<td>#13</td>
<td>2</td>
<td>0.48</td>
</tr>
<tr>
<td>#14</td>
<td>6</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td>Lane Parallel $P_i$</td>
<td></td>
</tr>
<tr>
<td>#15</td>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td>#16</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>#17</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Shabal Final Additions in $P$</td>
<td></td>
</tr>
<tr>
<td>#18</td>
<td>Series</td>
<td>0.58</td>
</tr>
<tr>
<td>#19</td>
<td>Parallel</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>S-Hash</td>
<td></td>
</tr>
<tr>
<td>#20</td>
<td>Low Area</td>
<td>0.02</td>
</tr>
<tr>
<td>#21</td>
<td>Single Cycle</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Design T ype TP/Area

Area (slices)

Throughput (Gbps)
Virtex-5 Extended Graph

<table>
<thead>
<tr>
<th>Design</th>
<th>Type</th>
<th>TP/Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane</td>
<td>Parallel P_i</td>
<td></td>
</tr>
<tr>
<td>#12</td>
<td>1</td>
<td>0.40</td>
</tr>
<tr>
<td>#13</td>
<td>2</td>
<td>0.48</td>
</tr>
<tr>
<td>#14</td>
<td>6</td>
<td>0.65</td>
</tr>
<tr>
<td>Lane</td>
<td>Parallel P_i</td>
<td></td>
</tr>
<tr>
<td>#15</td>
<td>1</td>
<td>0.67</td>
</tr>
<tr>
<td>#16</td>
<td>2</td>
<td>84</td>
</tr>
<tr>
<td>#17</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>Shabal</td>
<td>Final Additions in P</td>
<td></td>
</tr>
<tr>
<td>#18</td>
<td>Series</td>
<td>0.58</td>
</tr>
<tr>
<td>#19</td>
<td>Parallel</td>
<td>0.52</td>
</tr>
<tr>
<td>S-Hash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#20</td>
<td>Low Area</td>
<td>0.02</td>
</tr>
<tr>
<td>#21</td>
<td>Single Cycle</td>
<td>0.94</td>
</tr>
</tbody>
</table>
Conclusions

- The 224/256-bit best throughput per slice on Spartan-3:
- **#5. Grøstl-224/256**, with S-boxes implemented in BRAM and the $P$ and $Q$ permutations in parallel ($0.75 \text{Mbps/slice}$)
Conclusions

- The 224/256-bit best throughput per slice on Spartan-3:
  - **#5. Grøstl-224/256**, with S-boxes implemented in BRAM and the $P$ and $Q$ permutations in parallel (0.75 Mbps/slice)

- The 384/512-bit best throughput per slice on Spartan-3:
  - **#8. Grøstl-384/512**, with S-boxes implemented in BRAM and the $P$ and $Q$ permutations interleaved (0.46 Mbps/slice)
Conclusions

- The 224/256-bit best throughput per slice on Virtex-5:
- #4. Grøstl-224/256, with S-boxes implemented in BRAM and the $P$ and $Q$ permutations interleaved
  $(2.01\, Mbps/slice)$
Conclusions

- The 224/256-bit best throughput per slice on Virtex-5:
  - #4. Grøstl-224/256, with S-boxes implemented in BRAM and the $P$ and $Q$ permutations interleaved
    ($2.01 \text{Mbps/slice}$)

- The 384/512-bit best throughput per slice on Virtex-5:
  - #21. single-cycle Spectral Hash ($0.94 \text{Mbps/slice}$)
Conclusions

Of the five hash functions studied, the **Grøstl** implementations currently give the best overall balance between throughput and area, when implemented on FPGAs.
Thank you for your time
Any Questions?

brianb, andrewb, markh, neilh, robertmce, weibop, liam@eleceng.ucc.ie