HOST 2026 Tutorial:
Part II
Lightweight Cryptography using Composite Mersenne Product Registers
May 4, 2026

*Arman Allahverdi and *^Vincent John Mooney III

^Associate Professor, *School of Electrical and Computer Engineering

^Adjunct Associate Professor, School of Computer Science

Georgia Institute of Technology

Atlanta, Georgia

Georgia Tech

Acknowledgement

- This work has been partially supported by the U.S. Department of Energy (DoE) Office of Cybersecurity, Energy Security, and Emergency Response (CESER) under Cybersecurity for Energy Delivery Systems (CEDS) Agreement Number #DE-CR0000055 to the Georgia Tech Research Corporation: GRIDLOGIC: Hardware/Software Codesign for Deep Grid Visibility and Security

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Introduction

- Arman Allahverdi
  - B.S. in Electrical Engineering from the University of New Mexico, Albuquerque, New Mexico, 2021
  - M.S. in Electrical and Computer Engineering from The Georgia Institute of Technology, Atlanta, Georgia, 2024
  - 4th-year Ph.D. student in Electrical and Computer Engineering (Hardware-Software Codesign for Security) at The Georgia Institute of Technology

Introduction

- Vincent John Mooney III
  - B.S. in Electrical Engineering, Yale University, 1991
  - B.S. in Computer Science, Yale University, 1991
  - M.S. In Electrical Engineering, Stanford University, 1994
  - M.A. in Philosophy, 1997
  - Ph.D. in Electrical Engineering, 1998
  - Associate Professor, School of Electrical and Computer Engineering, The Georgia Institute of Technology
  - Adjunct Associate Professor, School of Computer Science, The Georgia Institute of Technology

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Terminology and Notation

- **Mersenne Exponent:** An integer n such that $2^n - 1$ is prime.
- **Mersenne Prime:** A prime number in the form $2^n - 1$.
- **Feedback Register:** A form of register where the next state, $A[t+1]$, is governed by a function f of the current state $A[t]$ (e.g., $A[t+1] = f(A[t])$.
- **Feedback Shift Register:** A form of feedback register where the output of each state bit a_i is connected to the input of the next bit a_{i+1}.
- **Linear Feedback Shift Register (LFSR):** A feedback shift register where successive states are determined by a linear function of the current state.
- **Nonlinear Feedback Shift Register (NLFSR):** A feedback shift register where successive states are determined by a nonlinear function of the current state.
- **Feedback Polynomial:** A polynomial that determines the feedback taps of a feedback register.
- **Primitive Polynomial:** An irreducible (unfactorable) polynomial of degree n over a finite field $GF(p)$ whose roots are primitive elements of the extension field $GF(p^m)$.

Terminology and Notation

- **Product Register:** A feedback register that performs Galois Field Multiplication of the current state $A[t]$ and an update polynomial U, modulo a feedback polynomial P.
- **Mersenne Product Register (MPR):** A product register of size n, where n is a Mersenne exponent.
- **Composite Mersenne Product Register (CMPR):** A feedback register formed by several MPRs combined through nonlinear chaining functions.
- **Stream Cipher:** An encryption scheme that encrypts data by performing an XOR with a pseudorandom bitstream.
- **Hash Function:** A one-way mapping, either keyed or unkeyed, of an arbitrary-length bit sequence to a fixed-length bit sequence.
- **Keystream:** A pseudorandom bitstream generated from a secret key and a public initialization vector (IV).
- **Chaining Density:** The proportion of the bits of a CMPR that receive chaining functions.
- **Initialization Round:** A single clocking of a feedback register (application of the next state function to the feedback register).
- **Authenticated Encryption with Associated Data (AEAD):** A cryptographic mode of operation that simultaneously ensures data confidentiality, integrity, and authenticity, with support for the transmission of authenticated (but unencrypted) data
- **Algebraic Degree (of an expression):** The largest number of variables in the product terms of an expression

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

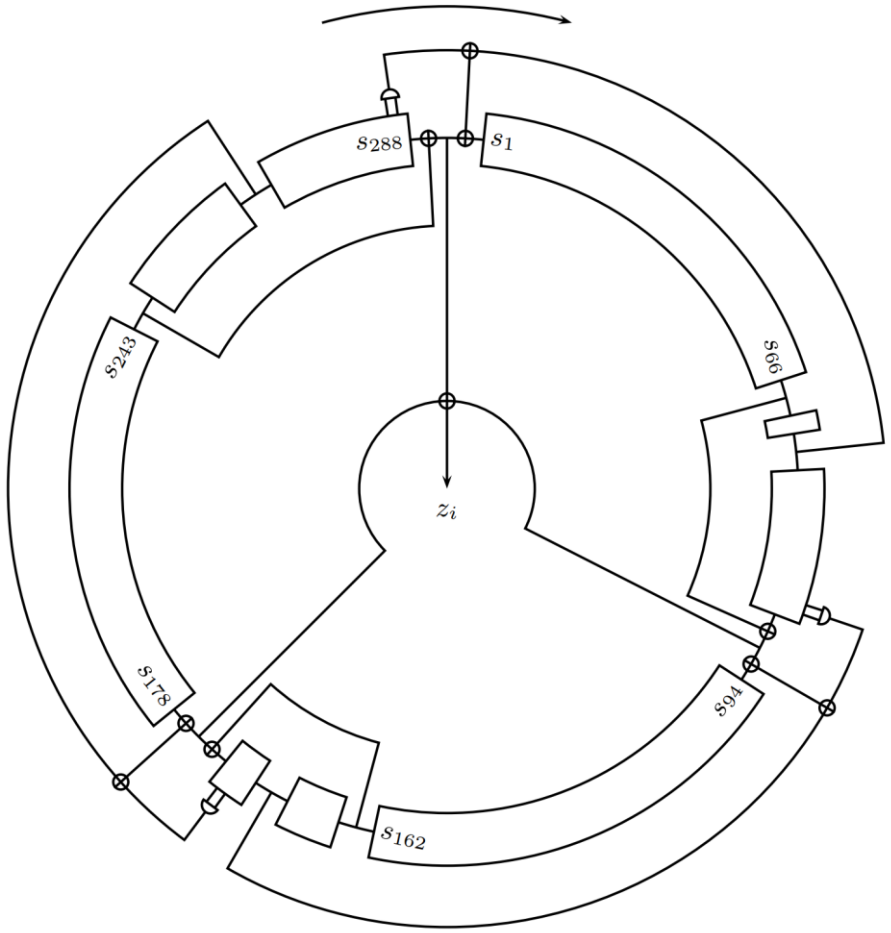
Prior Work: Register-Based Stream Ciphers

- Throughout this talk, we will refer to various register-based stream ciphers from the literature
- We concern ourselves with the following stream ciphers:
  - Trivium [9]
  - Grain-128AEADv2 [6]
  - Espresso [10]
- Capabilities and security claims:

Design	Register Type	Authentication	Key Size	IV Size	Security Claim
Trivium	Composed NLFSRs	No	80 bits	80 bits	2^{80}
Grain-128AEADv2	NLFSR LFSR	Yes	128 bits	96 bits	2^{128} (Key) 2^{64} (Tag)
Espresso	NLFSR	No	128 bits	96 bits	2^{128}

Prior Work: Trivium

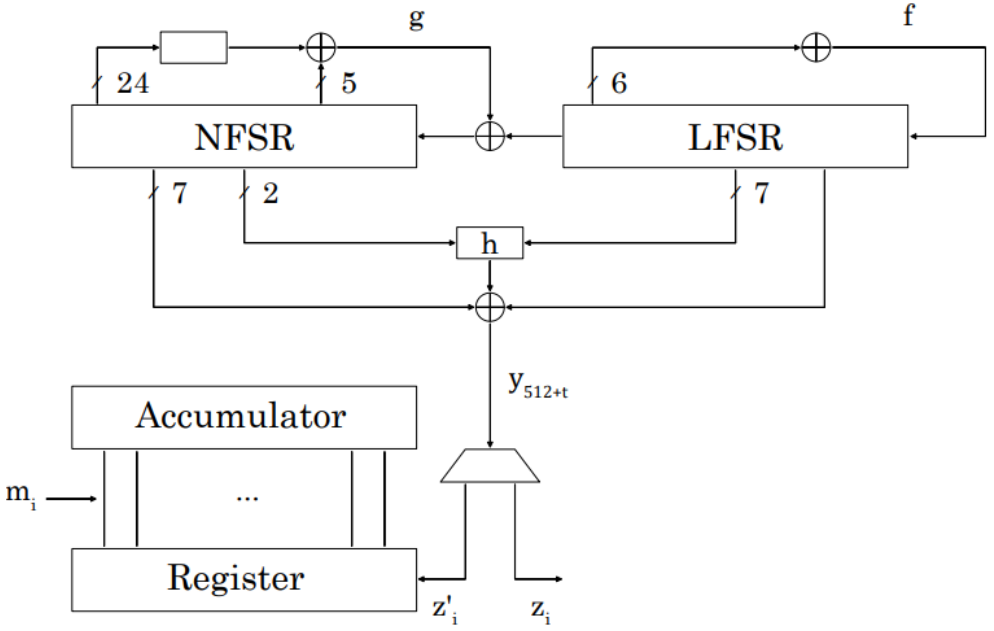
- Trivium is a stream cipher based on a 288-bit composite feedback register formed by the interconnection of three NLFSRs
- **Parameter Sizes:** 80-bit key, 80-bit IV
- **Initialization:** 1152 clock cycles applied to the 288-bit feedback register
- Often referred to as the most compact secure stream cipher, requiring ~3488 NAND gates to implement [9]



Visualization of the 288-bit Trivium Feedback Register [9]

Prior Work: Grain-128AEADv2

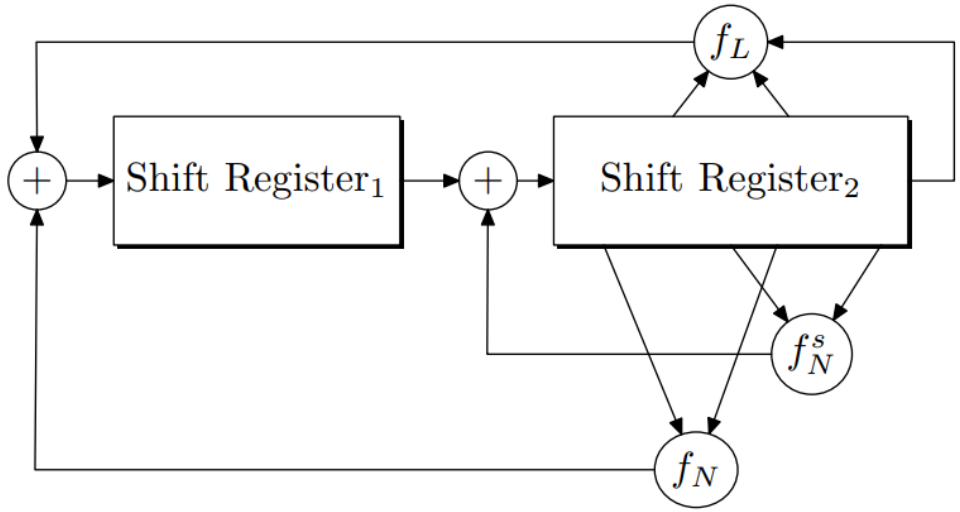
- Grain-128AEADv2 is an AEAD-capable stream cipher based on the interconnection of a 128-bit NLFSR and 128-bit LFSR
- Authentication provided by a keyed hash function instantiated from an accumulator and shift register
- One of the 10 finalists in the NIST LWC
- **Parameter Sizes:** 128-bit key, 96-bit IV, 64-bit tag
- **Initialization:** 512 clock cycles applied to the LFSR and NLFSR, with the key and IV periodically XORed into the NLFSR and LFSR, respectively



The Grain-128AEADv2 Architecture [6]

Prior Work: Espresso

- Espresso is a stream cipher based on a 256-bit NLFSR, where the NLFSR is generated using the notion of cross-joined pairs [10]
- **Parameter Sizes:** 128-bit key, 96-bit IV
- **Initialization:** 259 clock cycles applied to the NLFSR
- Geared toward 5G wireless communication systems



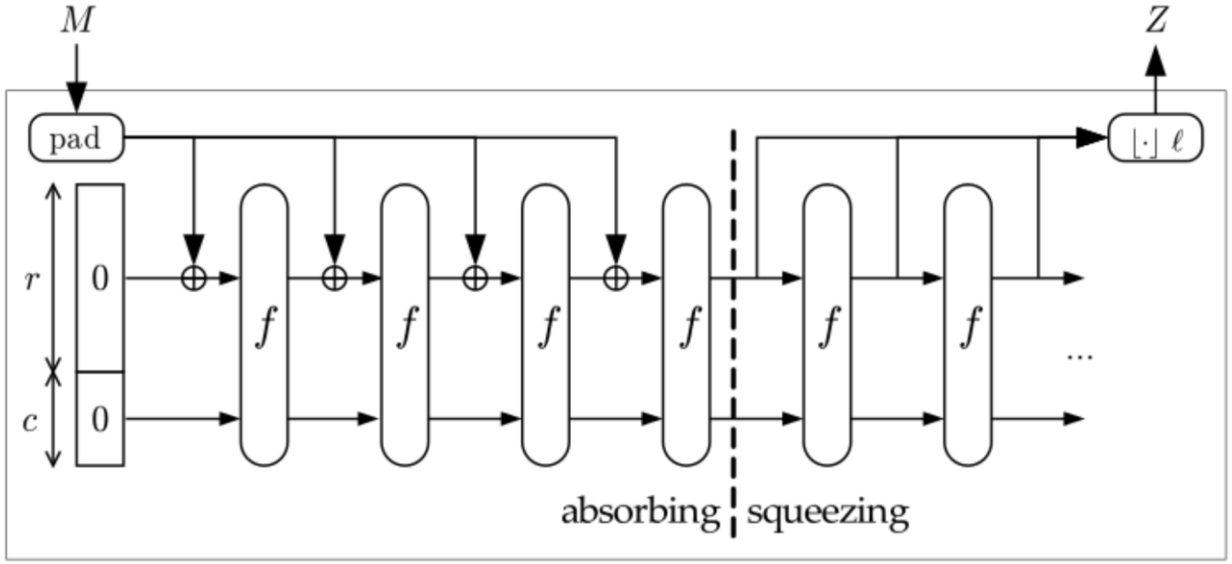
The Espresso NLFSR [10]

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Prior Work: The Sponge Construction

- The Sponge Construction [14] is a cryptographic design paradigm that provides a framework for turning a finite-sized permutation f into a cryptosystem with mathematically-proven security claims
- Used by the NIST-standardized hashing algorithm SHA-3 [25] and NIST LWC winner Ascon [16]

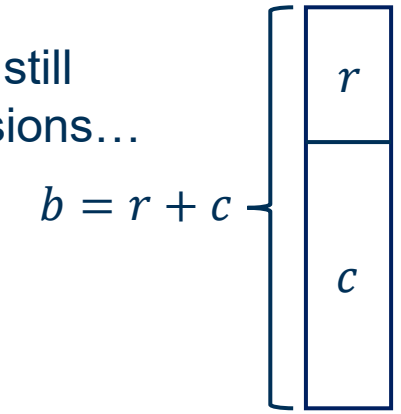


The Sponge Construction with a Generic Permutation f

Prior Work: The Sponge Construction

- The Sponge Construction partitions the state of a finite-sized permutation f as follows:
  - **Capacity:** c, no direct access permitted
  - **Rate:** r, direct access (e.g., XORing in data) allowed
  - **Total State Size:** $b = r + c$
- The Flat Sponge Claim provides a (collision resistance) security claim of $2^{c/2}$ [14, 15]
  - Factor of ½ comes from the Birthday Attack
- **Advantage:** Using the Sponge Construction reduces the security analysis to that of f*

*Some amount of statistical testing should still take place to search for internal state collisions…

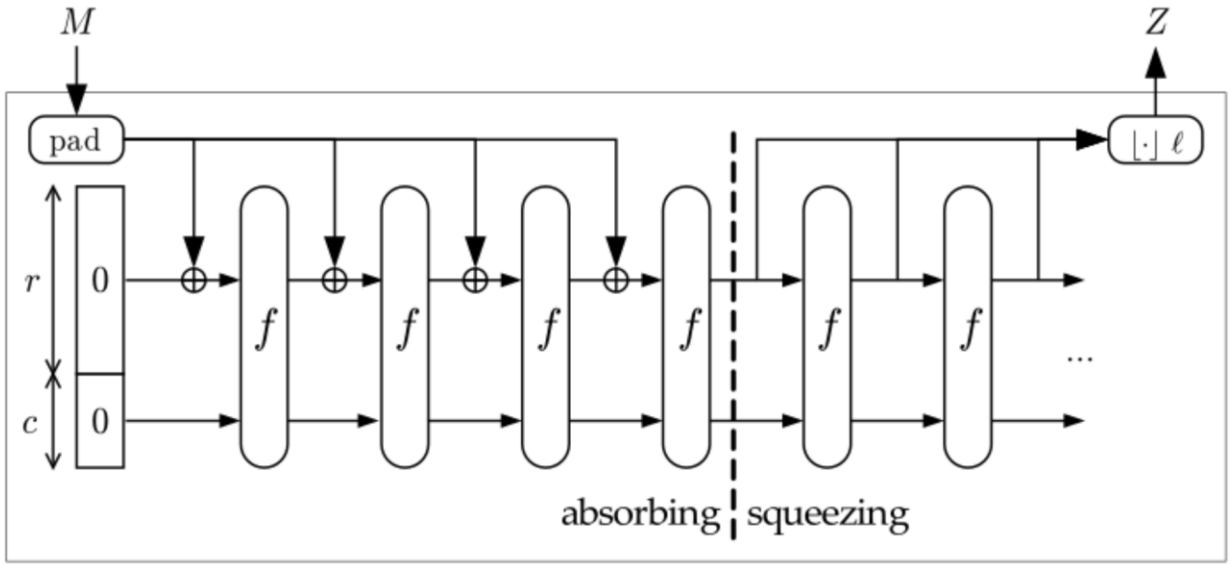


Example: $r = 64$, $c = 256$
$\Rightarrow b = 320$
$\Rightarrow 2^{c/2} = 2^{128}$

Sponge Construction State Partitioning

Prior Work: The Sponge Construction

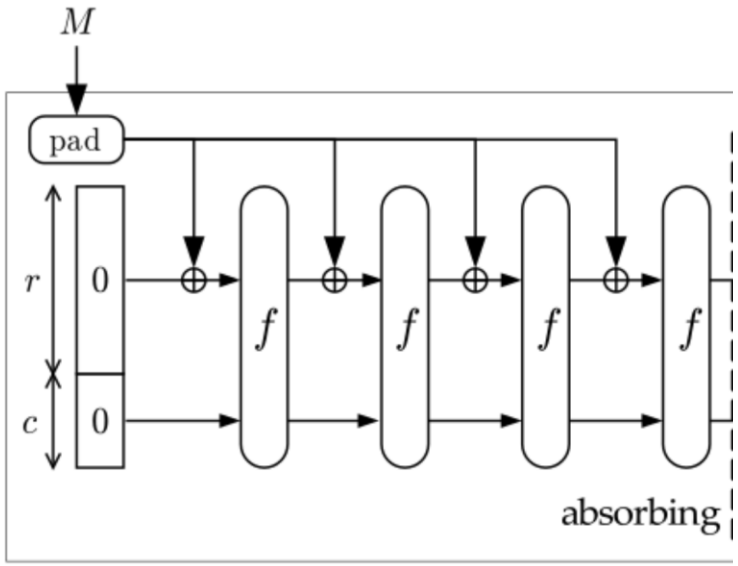
- The generic Sponge Construction utilizes two distinct phases for processing data and generating an output:
  - **Absorption:** The input is XORed with the rate portion of f
  - **Squeezing:** The output is extracted from the rate portion of f



The Sponge Construction with a Generic Permutation f [14]

Prior Work: The Sponge Construction

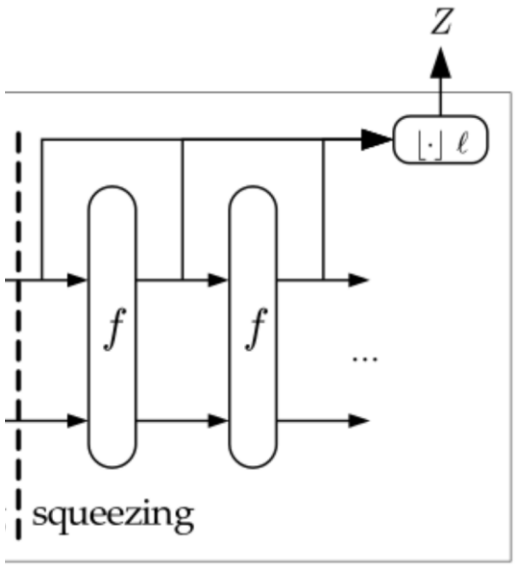
- The generic Sponge Construction utilizes two distinct phases for processing data and generating an output:
  - **Absorption:** The input is XORed with the rate portion of f



The Sponge Construction with a Generic Permutation f [14]

Prior Work: The Sponge Construction

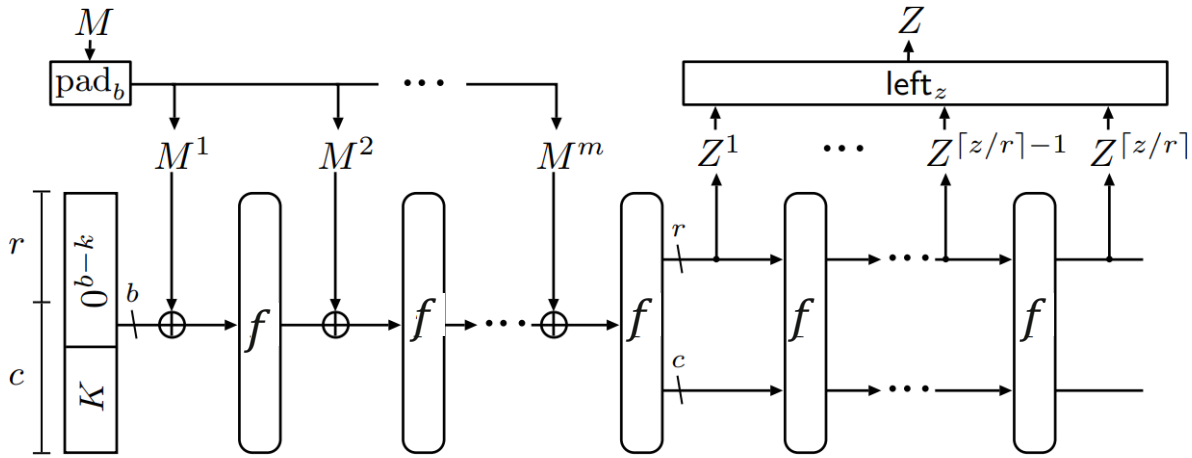
- The generic Sponge Construction utilizes two distinct phases for processing data and generating an output:
  -
  - **Squeezing:** The output is extracted from the rate portion of f



The Sponge Construction with a Generic Permutation f [14]

Prior Work: The Sponge Construction

- The Sponge Construction can be used with a secret key
- A popular mode of operation for keyed Sponge Constructions is the Full-State Keyed Sponge (FKS) [15]
- For a k-bit secret key and a c-bit capacity, FKS provides a security level of $2^{\min(k,c/2)}$ [15]
- FKS modifies the absorption phase by allowing the input to be XORed with the entire state of f rather than solely the rate portion (good for low-latency applications)



The FKS Mode of Operation [15]

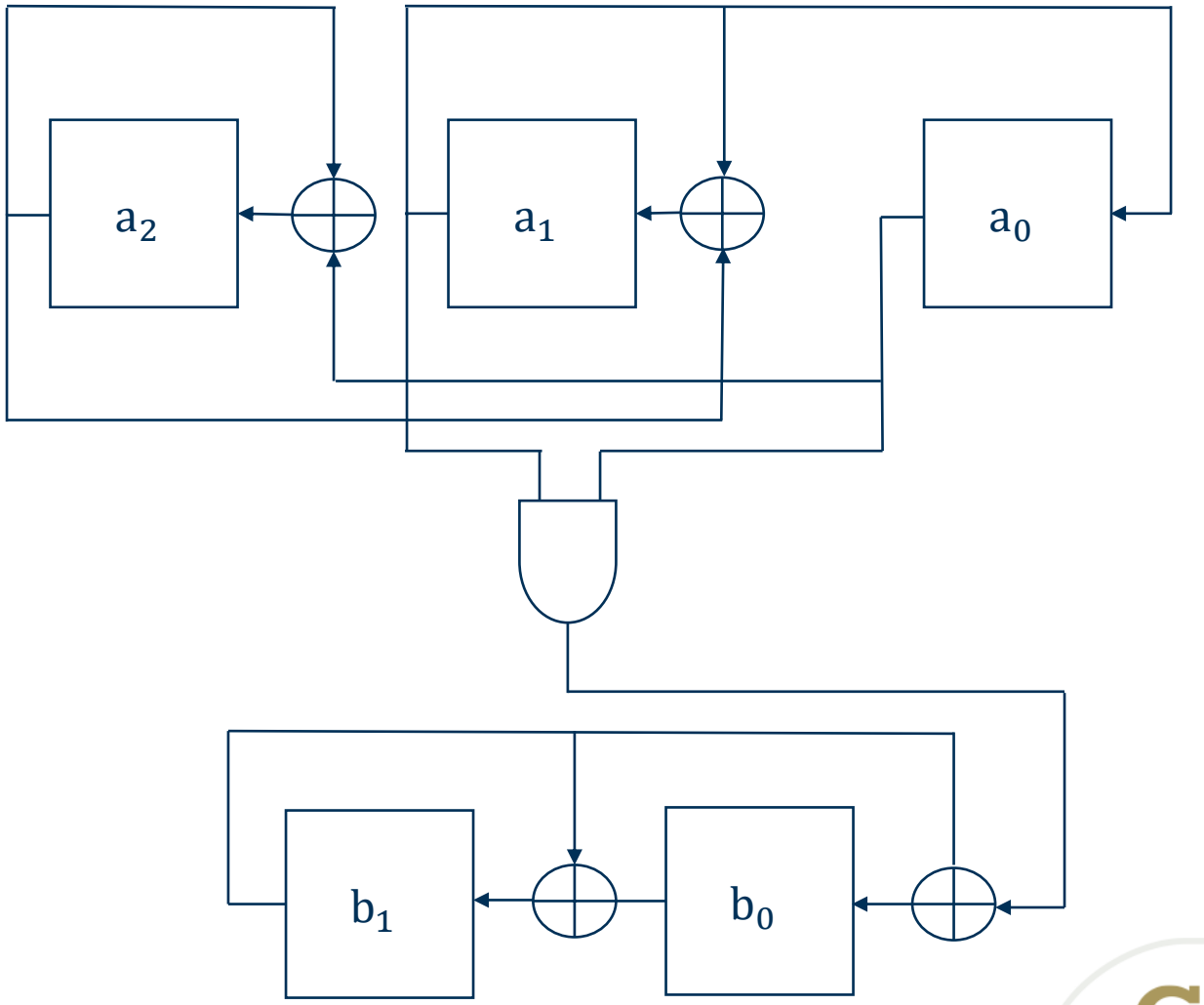
Georgia Tech

Outline

- Introduction
- Terminology and Notation
- Prior Work
  - Register-Based Stream Ciphers
  - The Sponge Construction
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

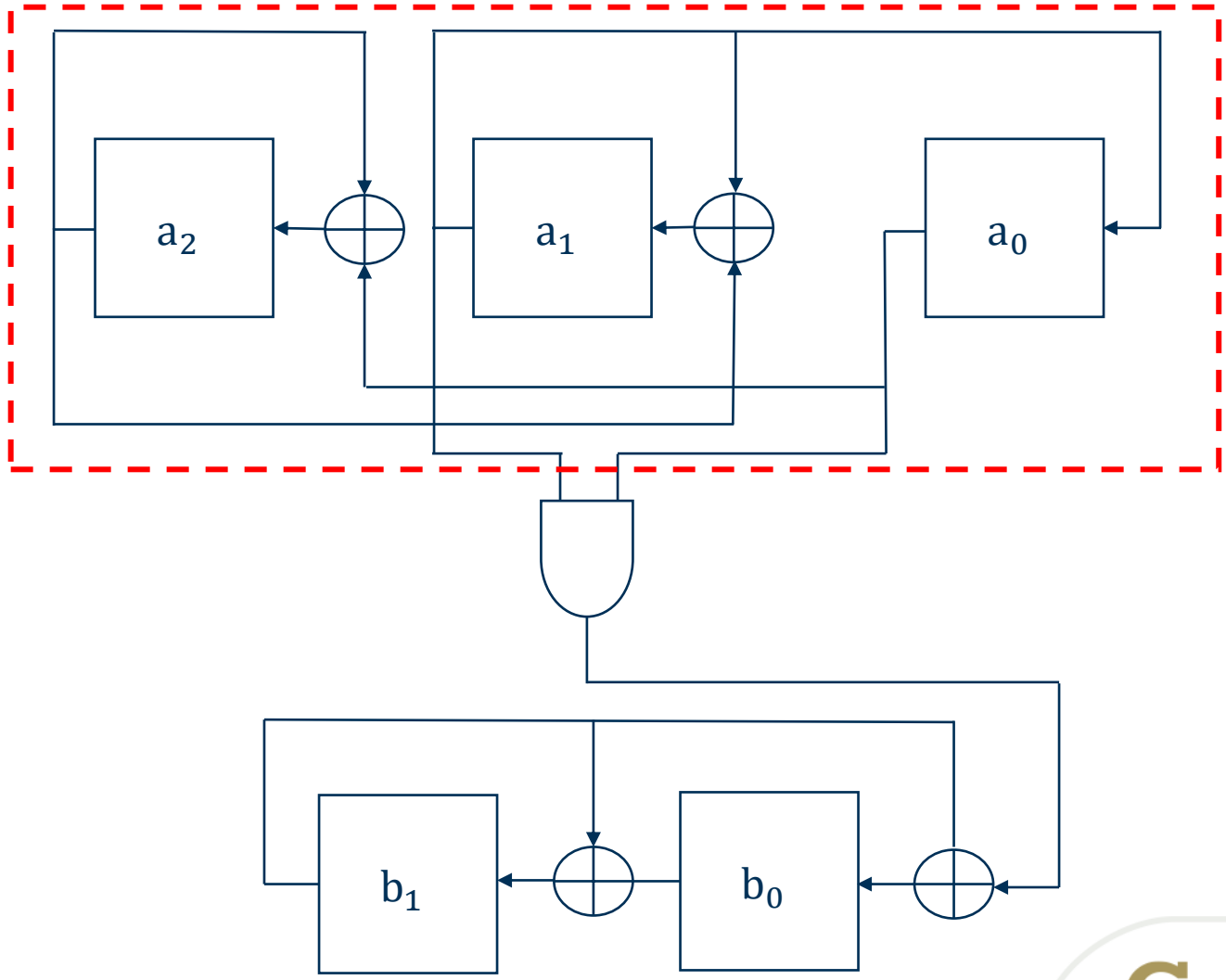
Lightweight Cryptography using CMPRs: Overview

- To demonstrate the cryptographic applications of CMPRs, we have designed several cryptosystems that use CMPRs as their source of (pseudo-)randomness
- We believe that CMPRs can serve as a self-contained cryptographic primitive containing both linear mixing (diffusion) layers and nonlinear (confusion) layers
- **Illustrative Small Example:**
  - 5-bit CMPR: 3-bit MPR + 2-bit MPR
  - Linear Mixing: 3-bit MPR
  - Nonlinearity: the chaining function, and the 2-bit MPR “perturbed” by the chaining function



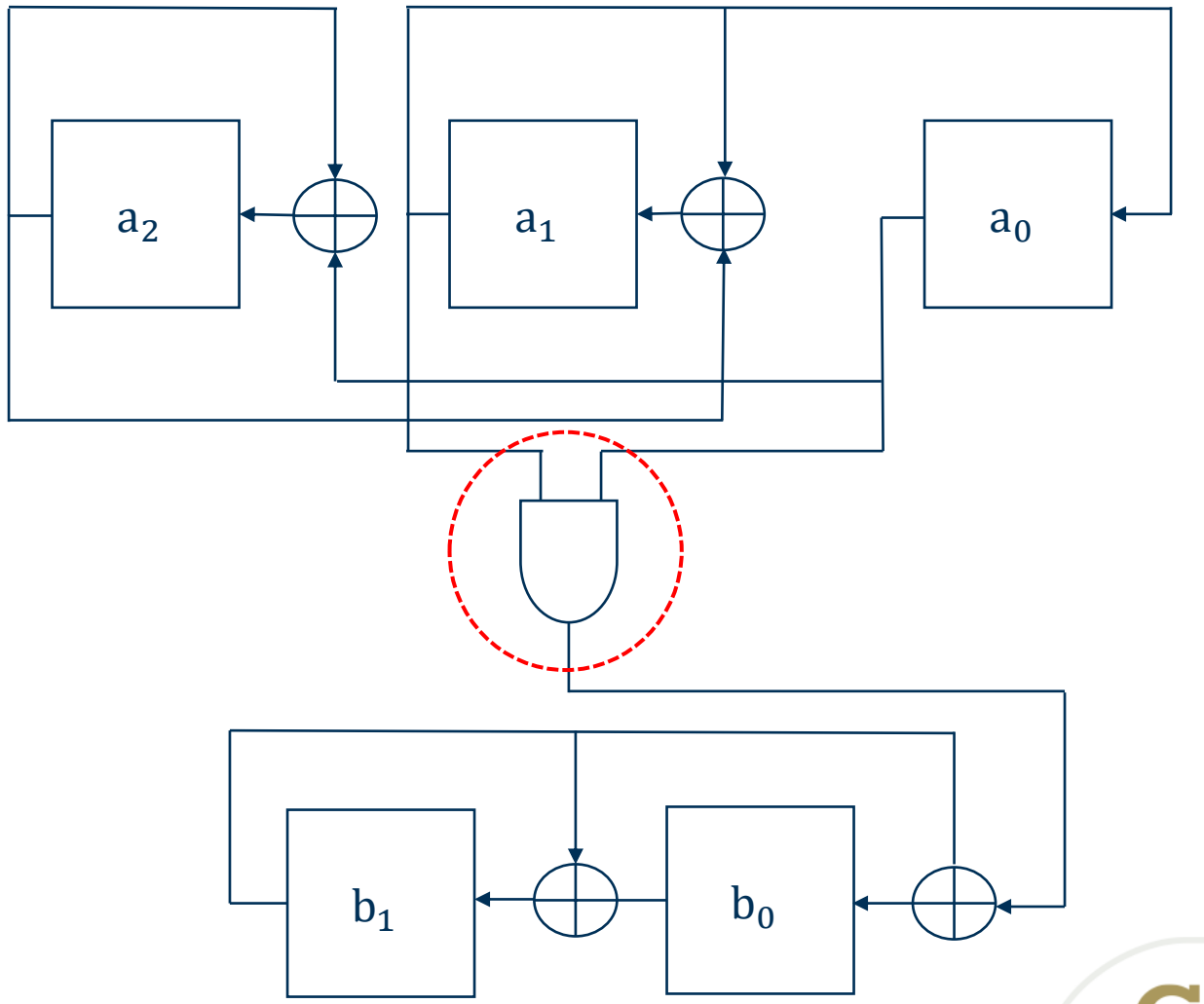
Lightweight Cryptography using CMPRs: Overview

- To demonstrate the cryptographic applications of CMPRs, we have designed several cryptosystems that use CMPRs as their source of (pseudo-)randomness
- We believe that CMPRs can serve as a self-contained cryptographic primitive containing both **linear mixing (diffusion)** layers and nonlinear (confusion) layers
- **Illustrative Small Example:**
  - 5-bit CMPR: 3-bit MPR + 2-bit MPR
  - Linear Mixing: 3-bit MPR
  - Nonlinearity: the chaining function, and the 2-bit MPR “perturbed” by the chaining function



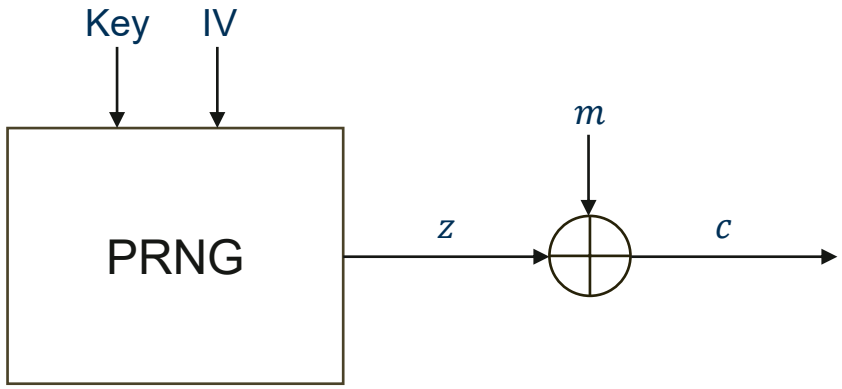
Lightweight Cryptography using CMPRs: Overview

- To demonstrate the cryptographic applications of CMPRs, we have designed several cryptosystems that use CMPRs as their source of (pseudo-)randomness
- We believe that CMPRs can serve as a self-contained cryptographic primitive containing both linear mixing (diffusion) layers and **nonlinear (confusion)** layers
- **Illustrative Small Example:**
  - 5-bit CMPR: 3-bit MPR + 2-bit MPR
  - Linear Mixing: 3-bit MPR
  - Nonlinearity: the chaining function, and the 2-bit MPR “perturbed” by the chaining function



Stream Cipher Design: Background

- Initial research on PR-based cryptography focused on stream cipher design
- Stream ciphers are essentially keyed, cryptographically-secure PRNGs that generate a variable-length pseudorandom keystream z
- Ciphertext c is generated from the message m according to $c = m \oplus z$ ("additive" stream cipher)



A Generic Additive Stream Cipher

Stream Cipher Design: Background

- Stream cipher design paradigms:
  - **Bit-Oriented:** Cipher utilizes bitwise operations and generates keystream at 1 bit/cycle
  - **Word-Oriented:** Cipher utilizes byte-level operations and generates bytes of keystream/cycle
- Stream ciphers based on feedback registers, including the previously discussed examples from the prior work, are typically bit-oriented for the following reasons:
  - Bitwise operations are inexpensive in hardware
  - Feedback register-based ciphers have a finite, one-dimensional internal state
- Bit-oriented stream ciphers sacrifice throughput for lower latency and a more efficient hardware implementation
- The CMPR-based stream ciphers that will be discussed in this presentation are also bit-oriented

Stream Cipher Design: Background

- Our first pass at designing CMPR-based stream ciphers resulted in a family of three CMPR-based stream ciphers
  - Each of the three ciphers utilizes a different CMPR along with a different key and IV size
- We use the Trivium stream cipher as a point of comparison with respect to the CMPR-based stream cipher family
- Trivium is a great comparison baseline because:
  - Design is based on a 288-bit “composite feedback register” (similar overall design philosophy to CMPRs, but without the periodicity and linear complexity proofs)
  - Highly compact in hardware (colloquially, the smallest stream cipher)

CMPR-Based Stream Cipher Design

- We design our family of CMPR-based stream ciphers as follows:

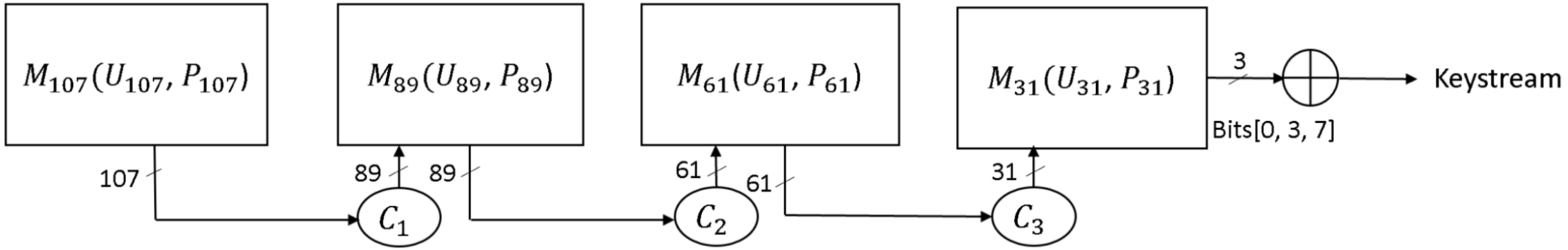
Version	CMPR Size	Key Size	IV Size
v1	288 bits	128 bits	128 bits
v2	162 bits	80 bits	80 bits
v3	170 bits	84 bits	84 bits

- IV size also varied to ensure support for equally-sized parameters, as in the case of Trivium
- All three stream ciphers operate according to the following procedure:
  - Key and IV stored in MSBs (key first)
  - CMPR clocked 100 times
    - At 50 rounds, the upper and lower halves of the state are swapped
  - On each clock cycle (post-100 initialization rounds), keystream bit generated by computing XOR of state bits 0, 3, and 7

CMPR-Based Stream Cipher Design

- **v1:** 288-bit CMPR, 128-bit key, 128-bit IV, same state size as Trivium

MPR Size	Update Polynomial	Primitive Polynomial
107 bits	$U_{107}(x) = x$	$P_{107}(x) = x^{107} + x^{89} + x^{84} + x^{40} + x^{29} + x^{23} + 1$
89 bits	$U_{89}(x) = x$	$P_{89}(x) = x^{89} + x^{81} + x^{68} + x^{31} + x^{21} + x^{18} + 1$
61 bits	$U_{61}(x) = x$	$P_{61}(x) = x^{61} + x^{44} + x^{19} + x^{15} + 1$
31 bits	$U_{31}(x) = x$	$P_{31}(x) = x^{31} + x^3 + x^2 + x + 1$

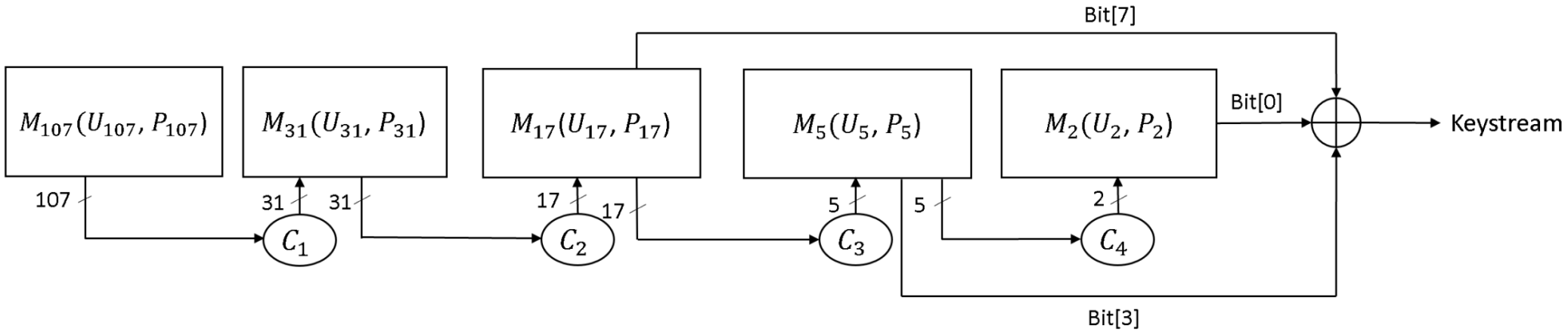


CMPR-Based Stream Cipher v1 [1]

CMPR-Based Stream Cipher Design

- **v2:** 162-bit CMPR, 80-bit key, 80-bit IV, same security claim as Trivium

MPR Size	Update Polynomial	Primitive Polynomial
107 bits	$U_{107}(x) = x$	$P_{107}(x) = x^{107} + x^{89} + x^{84} + x^{40} + x^{29} + x^{23} + 1$
31 bits	$U_{31}(x) = x$	$P_{31}(x) = x^{31} + x^3 + x^2 + x + 1$
17 bits	$U_{17}(x) = x$	$P_{17}(x) = x^{17} + x^8 + x^7 + x^6 + x^4 + x^3 + 1$
5 bits	$U_5(x) = x$	$P_5(x) = x^5 + x^4 + x^3 + x^2 + 1$
2 bits	$U_2(x) = x$	$P_2(x) = x^2 + x + 1$

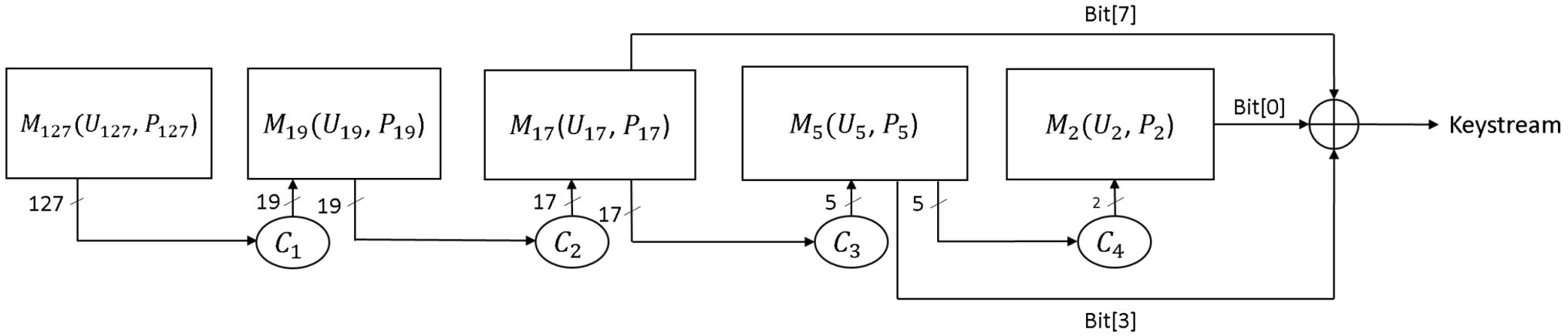


CMPR-Based Stream Cipher v2 [1]

CMPR-Based Stream Cipher Design

- **v3:** 170-bit CMPR, 84-bit key, 84-bit IV, slightly higher security claim than Trivium but smaller state size

MPR Size	Update Polynomial	Primitive Polynomial
127 bits	$U_{127}(x) = x$	$P_{127}(x) = x^{127} + x^{54} + x^{45} + x^{13} + 1$
19 bits	$U_{19}(x) = x$	$P_{19}(x) = x^{19} + x^5 + x^4 + x^3 + x^2 + x + 1$
17 bits	$U_{17}(x) = x$	$P_{17}(x) = x^{17} + x^8 + x^7 + x^6 + x^4 + x^3 + 1$
5 bits	$U_5(x) = x$	$P_5(x) = x^5 + x^4 + x^3 + x^2 + 1$
2 bits	$U_2(x) = x$	$P_2(x) = x^2 + x + 1$



CMPR-Based Stream Cipher v3 [1]

CMPR-Based Stream Cipher Design

- ASIC target (FreePDK45 45nm process [13]) implementation results, $F_{clk} = 300MHz$ [1]:

Cipher	Area (μm^2)	Power (mW)	Key Security	Init. Rounds
TRIVIUM	5627	3.0226	2^{80}	1152
CMPR stream cipher v_1	12909	5.4405	2^{128}	100
CMPR stream cipher v_2	7057	2.8894	2^{80}	100
CMPR stream cipher v_3	7292	2.8993	2^{84}	100

Cipher	Area Relative to TRIVIUM	Power Relative to TRIVIUM
CMPR stream cipher v_1	+129.4%	+80.0%
CMPR stream cipher v_2	+25.4%	-4.41%
CMPR stream cipher v_3	+29.6%	-4.08%

CMPR-Based Stream Cipher Design

- Higher area relative to Trivium is explained by:
  - **Boolean Logic Overhead:** CMPRs require a higher number of Boolean logic gates to implement chaining functions
  - **Swapping Overhead:** Large MUX inferred to facilitate state swap during initialization phase

Cipher	Area Relative to TRIVIUM	Power Relative to TRIVIUM
CMPR stream cipher v_1	+129.4%	+80.0%
CMPR stream cipher v_2	+25.4%	-4.41%
CMPR stream cipher v_3	+29.6%	-4.08%

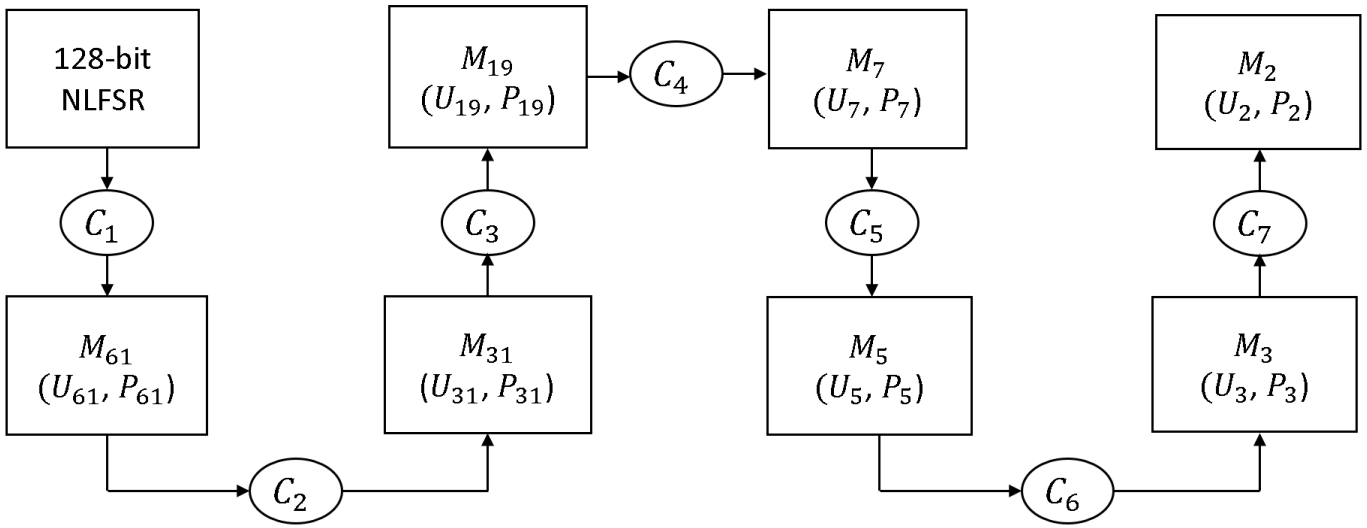
Outline

- Introduction
- Terminology and Notation
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Georgia Tech

Authenticated Hybrid Stream Cipher Design

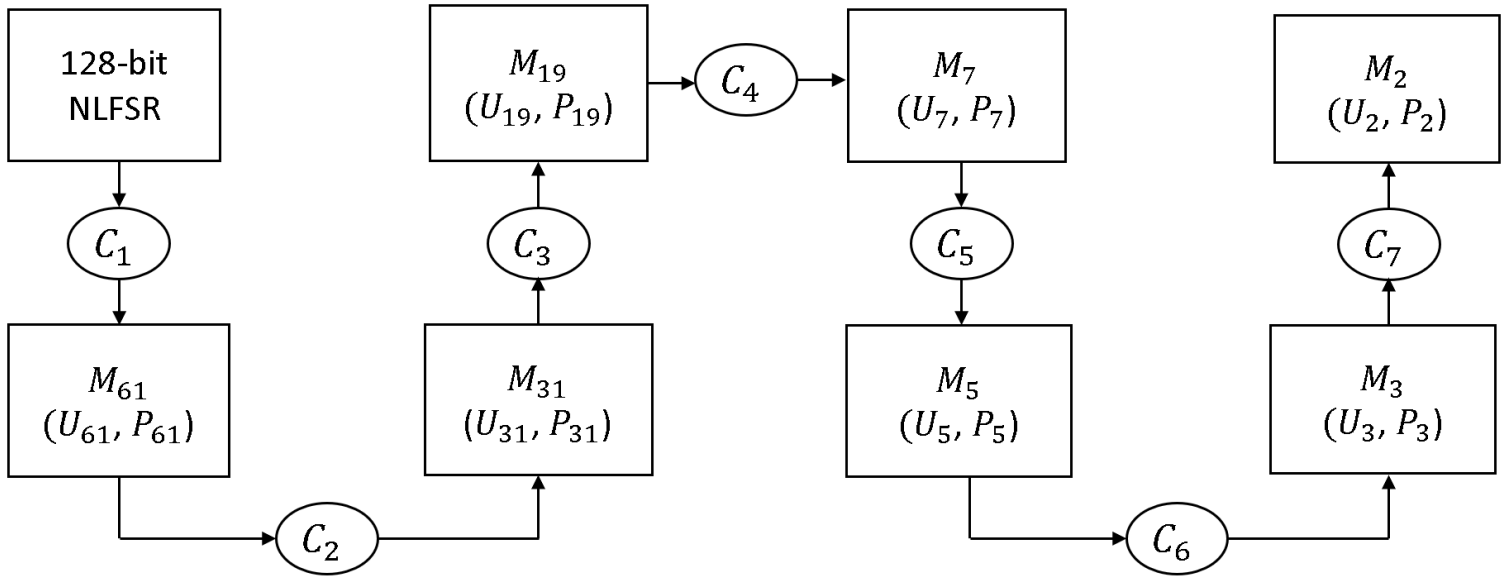
- Our next pass at CMPR-based encryption addressed areas left unexplored by our initial attempt:
  - Message authentication
  - Authenticated encryption with associated data (AEAD)
  - Broader set of comparisons to NLFSR-based stream ciphers
- We also sought to use MPRs and NLFSRs in conjunction with one another



256-bit Hybrid Register
C_i: **Sets of Chaining Functions**

Authenticated Hybrid Stream Cipher Design

- The Chaining Period Theorem and linear complexity analyses of [1] allow for the top-level (largest) register in a CMPR construction to be non-Mersenne (e.g., an LFSR, NLFSR, or generic Product Register), with one key condition:
  - Periodicity and linear complexity analyses of [1] will only apply to the set of subregisters with sizes coprime to one another (e.g., no common divisors besides 1)
- We leveraged this fact to create a 256-bit *hybrid register* [2]
  - 128-bit NLFSR chained into a 128-bit CMPR

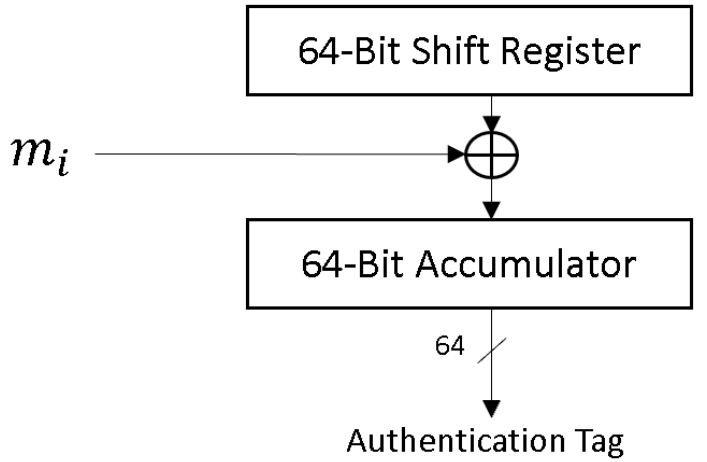


256-bit Hybrid Register
C_i: **Sets of Chaining Functions**

Georgia Tech

Authenticated Hybrid Stream Cipher Design

- **128-bit NLFSR:**
  - Sourced from Grain-128AEADv2 [6]
  - Nonlinear feedback function with algebraic degree 4
- **128-bit CMPR:**
  - Constructed using 7 MPRs of different sizes
  - Chaining functions with algebraic degree 4 between each pair of MPRs
- The 256-bit hybrid register formed by interconnecting the NLFSR and CMPR serves as the PRNG for the *hybrid register stream cipher*
- **Parameter Sizes:** 128-bit key, 96-bit IV
- **Authentication:**
  - Used the Grain-128AEADv2 authenticator
  - Security proof based on universal hash functions [8]



Grain-128AEADv2 Authenticator

Authenticated Hybrid Stream Cipher Design

- The hybrid register stream cipher operates according to the following initialization algorithm [2]:
  - **Part I: Keystream Generator Initialization**
    - Key and IV stored in MSBs (key first)
    - Hybrid register clocked 128 times
    - On each clock cycle, encryption bit generated by computing XOR of state bits 0, 3, and 7
  - We clock the hybrid register 128 times to ensure sufficient randomization in the NLFSR portion of the state:
    - 128 clock cycles ensures a “full-state rotation” of the 128-bit NLFSR, e.g., every state bit is processed by the nonlinear feedback function once

Authenticated Hybrid Stream Cipher Design

- **Part II: Authenticator Initialization**
  - Hybrid register clocked an additional 128 times
  - Accumulator and shift register filled with encryption (keystream) bits
  - This methodology means that our stream cipher follows the **encrypt-then-MAC** procedure
- Once both parts of the initialization phase are completed, the cipher operates as follows:
  - On every even clock cycle, the generated keystream bit is output as an encryption bit
  - On every odd clock cycle, the generated keystream bit is used as an authentication bit and stored in the MSB of the 64-bit shift register
  - Contents of 64-bit accumulator XORed with plaintext message bits and shift register bits
  - 64-bit accumulator state output as authentication tag

Authenticated Hybrid Stream Cipher Design

- In hardware, we compare the hybrid register stream cipher to the following register-based stream ciphers:
  - Grain-128AEADv2 [6]
  - Espresso [9]
  - Trivium [10]
- Security levels, authentication capabilities, and initialization latency for each cipher [2]:

Design	Authentication	Security
Hybrid Register	Yes	2^{128}
Grain-128AEADv2	Yes	2^{128}
Espresso	No	2^{128}
TRIVIUM	No	2^{80}

Design	Latency (Clock Cycles)
Hybrid Register	128
Grain-128AEADv2	512
Espresso	256
TRIVIUM	1152

- FreePDK45 45nm process [13], and $F_{clk} = 300MHz$
- For a fair comparison, we exclude the authentication hardware overhead from our comparison to Espresso and Trivium

Authenticated Hybrid Stream Cipher Design

- Comparison with respect to area, average power, and the power-delay product (PDP):
  - PDP = $P_{avg} * T_{clk}$
- Unauthenticated comparison vs. Trivium and Espresso:

Design	Area (μm^2)	Power (mW)	PDP (pJ)
Hybrid Register	4402	0.981	3.27
Espresso	5894	1.949	6.49
TRIVIUM	5627	3.0226	10.08

- **Throughput:** Since these are all bit-oriented stream ciphers synthesized at 300MHz, throughput is 300Mbps

Authenticated Hybrid Stream Cipher Design

- Comparison with respect to area, average power, and the power-delay product (PDP):
  - PDP = $P_{avg} * T_{clk}$
- Authenticated comparison vs. Grain-128AEADv2:

Design	Area (μm^2)	Power (mW)	PDP (pJ)
Hybrid Register	5650	1.313	4.37
Grain-128AEADv2	6401	2.936	9.78

- **Throughput:** Since these are all bit-oriented stream ciphers synthesized at 300MHz, but encryption bits are only produced on alternating clock cycles, throughput is 150Mbps

Authenticated Hybrid Stream Cipher Design

- Lower area of the hybrid register stream cipher is explained by:
  - **Simplistic Output Function and Initialization:** Grain-128AEADv2 and Espresso utilize complicated nonlinear output functions, along with more dynamic initialization that requires multiplexing
  - **Internal State Size:** Trivium utilizes a 288-bit internal state, our design requires 256 bits of internal state
- Lower energy consumption of the hybrid register stream cipher is attributed to:
  - **Low Round Count:** Only 256 clock cycles required for both keystream generator and authenticator initialization
  - **Simplistic Initialization:** Each initialization round is just a register clocking

Design	Area (μm^2)	Power (mW)	PDP (pJ)
Hybrid Register	4402	0.981	3.27
Espresso	5894	1.949	6.49
TRIVIUM	5627	3.0226	10.08

Design	Area (μm^2)	Power (mW)	PDP (pJ)
Hybrid Register	5650	1.313	4.37
Grain-128AEADv2	6401	2.936	9.78

Authenticated Hybrid Stream Cipher Design

- Relative to Unauthenticated CMPR Stream Cipher Family:
  - **Lower Area and Power:** Achieved by reducing the chaining function density and omitting state swap during initialization

Design	**Area (μm^2)**	**Power (mW)**	**PDP (pJ)**
Hybrid Register	4402	0.981	3.27
Espresso	5894	1.949	6.49
TRIVIUM	5627	3.0226	10.08

Cipher	Area (μm^2)	Power (mW)	Key Security	Init. Rounds
TRIVIUM	5627	3.0226	2^{80}	1152
CMPR stream cipher v_1	12909	5.4405	2^{128}	100
CMPR stream cipher v_2	7057	2.8894	2^{80}	100
CMPR stream cipher v_3	7292	2.8993	2^{84}	100

Outline

- Introduction
- Terminology and Notation
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Georgia Tech

Hash Function Design

- After a thorough exploration of CMPR-based stream cipher design, we sought to design a hash function (namely a message authentication code) using CMPRs
- We chose the Sponge Construction [14, 15] as our hash function design framework
- **Main Contribution:** proposing the use of per-MPR update polynomials $U(x)$ as the MAC key:
  - Reduced implementation area on reconfigurable hardware targets
  - Number of possible $U(x)$ is exponential with respect to MPR size
- **C**MPR-based **A**rea-efficient **S**ponge **H**ash (CASH)

Hash Function Design: CMPR Specification

- For the permutation of our Sponge-based MAC, we use the following 192-bit CMPR:

MPR Size	U(x)	P(x)
107 bits	105-bit key fragment	$x^{107} + x^{59} + x^{54} + x^{39} + 1$
61 bits	23-bit key fragment	$x^{61} + x^{44} + x^{19} + x^{15} + 1$
19 bits	$x^{17} + 1$	$x^{19} + x^5 + x^2 + x + 1$
3 bits	$x^2 + x$	$x^3 + x + 1$
2 bits	$x + 1$	$x^2 + x + 1$

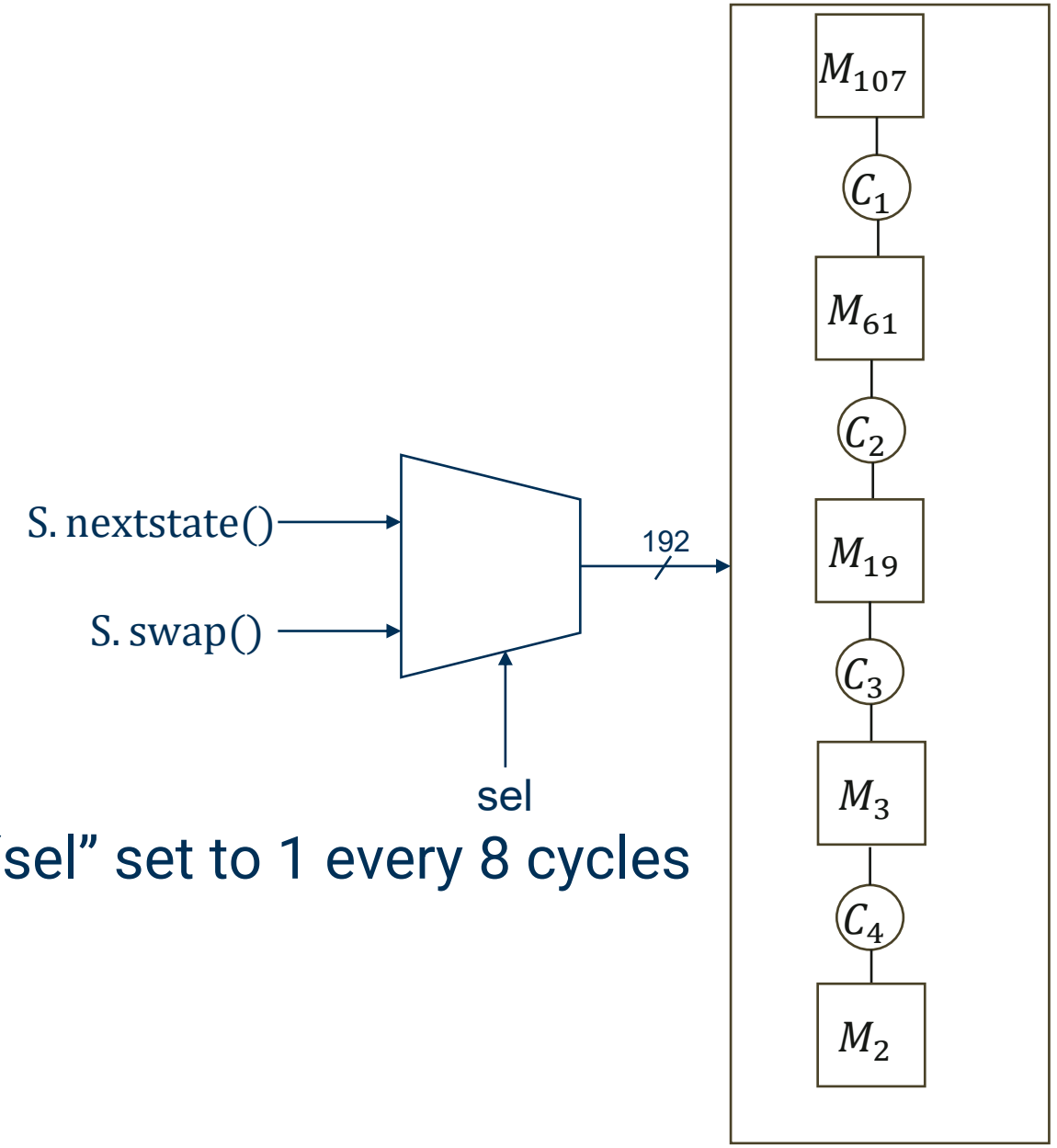
- A total of 105+23=128 bits of the update polynomials of the two largest MPRs are used as key bits:
  - **107-bit MPR:** 2 LSBs of $U(x)$ set to 10, all other bits allowed to vary
  - **61-bit MPR:** 23 MSBs allowed to vary, 2 LSBs set to 10, all other bits set to 0's
  - This is done to ensure $U(x) \neq 0,1$ for the 107- and 61-bit MPRs, irrespective of the key value chosen ($U(x) = 11 = x + 1$ is allowed, however)

Hash Function Design: Notation

- CASH algorithm-specific notation:
  - S: Internal state of the 192-bit CMPR
  - $S.\mathrm{nextstate}()$: State update function of the 192-bit CMPR
  - $S.\mathrm{swap}()$: Swap the upper and lower halves of S, preserving bit ordering
  - j: Number of message blocks to be processed
  - 1^i: An i-bit bitstream of 1's
  - k: A secret key
  - M_i: A 192-bit message block
  - M: A variable-length message
  - H_i: A 64-bit digest (hash) block
  - H: A 256-bit digest (hash)
  - *: A quantity of arbitrary length
  - ||: Concatenation

Hash Function Design: CASH Permutation

- 32-cycle CASH permutation with periodic swapping of the internal state halves
- Chaining functions C_i: at most 4-input AND, 4-input XOR
- **Chaining Density:** 50%
- **Sponge Parameters:** $r = 64, c = 128$
- **MPR Count:**
  - 5 MPRs
  - Chosen to ensure 4 chaining stages
  - Specific Mersenne exponent selection somewhat arbitrary, multiple configurations exist for 192 bits



- "sel" set to 1 every 8 cycles

Hash Function Design: Permutation Algorithm

- CASH permutation algorithm:
  - 32 clock cycles
  - 3 state swaps

Algorithm 1 CASH Permutation Algorithm

```
1:  procedure PERMUTE
2:      for i ← 0...7 do
3:          S.nextstate()
4:      end for
5:      S.swap()
6:      for i ← 0...7 do
7:          S.nextstate()
8:      end for
9:      S.swap()
10:     for i ← 0...7 do
11:         S.nextstate()
12:     end for
13:     S.swap()
14:     for i ← 0...7 do
15:         S.nextstate()
16:     end for
17: end procedure
```

Hash Function Design: Digest Generation Algorithm

- CASH hash generation algorithm:
  - (256-bit hash)
  - Message input and digest extraction performed according to Sponge Construction rules

Algorithm 2 CASH Algorithm

procedure INITIALIZE
  $S \leftarrow 1^{192}$
  $S.permute()$
end procedure
procedure ABSORB
  $M_0, \ldots, M_{j-1} \leftarrow M \| 1 \| 0^*$ ▷ Sponge Padding
  for $i \leftarrow 0 \ldots j-1$ **do**
    $S \leftarrow S \oplus M_i$
    $S.permute()$
  end for
end procedure
procedure SQUEEZE
  for $i \leftarrow 0 \ldots 3$ **do**
    $S.permute()$
    $H_i \leftarrow S(63), \ldots, S(0)$
  end for
end procedure
$H \leftarrow H_0 \| H_1 \| H_2 \| H_3$

Hash Function Design: Security Claims and Comparisons

- CASH is designed exclusively for reconfigurable targets (e.g., FPGAs, embedded FPGAs)
  - Changes to the key require resynthesis
- On an FPGA target, we perform a resource utilization comparison of the following designs, which all provide (roughly) similar security levels:

Design	Collision Resistance	(Second) Preimage Resistance
CASH	2^{128}	2^{128}
ASCON	2^{128}	2^{128}
Xoodyak	2^{128}	2^{128}
PHOTON-BEETLE	2^{112}	2^{128}
Romulus	2^{121}	2^{121}

Security Claims for CASH and Selected NIST Lightweight Cryptography Competition (2018-2023) Finalist Algorithms [16, 17, 18, 19]

- All designs synthesized onto the Intel DE10-Standard board [20]

Hash Function Design: FPGA Hardware Implementation

- FPGA hardware implementation results:
  - ALMs refer to adaptive logic modules, which are the combinational logic block element in the Intel FPGA suite
  - Area-delay product (ADP): $T_{clk} * ALMs$
  - Efficiency η: $Throughput/ALMs$

Design	ALMs	F_{max} (MHz)	Throughput (Mbps)
CASH	222	289	578.0
ASCON	400	239	1274.7
Xoodyak	853	218	2325.3
PHOTON-Beetle	892	242	645.3
Romulus	453	301	963.2

Design	Latency (Cycles)	ADP (ALM · ns)	η (Mbps/ALM)
CASH	32	768.17	2.60
ASCON	12	1673.64	3.19
Xoodyak	12	3912.84	2.73
PHOTON-Beetle	12	3685.95	0.72
Romulus	40	1504.98	2.13

  - CASH outperforms all selected baselines with respect to area utilization

Hash Function Design: FPGA Hardware Implementation

- **CASH Area Reduction:**
  - Reduced internal state size since key is embedded in $U(x)$
  - Security claim based on a CMPR invertibility theorem and exponential $U(x)$ space, not solely Sponge Construction

Design	ALMs	F_{max} (MHz)	Throughput (Mbps)
CASH	222	289	578.0
ASCON	400	239	1274.7
Xoodyak	853	218	2325.3
PHOTON-Beetle	892	242	645.3
Romulus	453	301	963.2

Design	Latency (Cycles)	ADP (ALM · ns)	η (Mbps/ALM)
CASH	32	768.17	2.60
ASCON	12	1673.64	3.19
Xoodyak	12	3912.84	2.73
PHOTON-Beetle	12	3685.95	0.72
Romulus	40	1504.98	2.13

  - CASH outperforms all selected baselines with respect to area utilization

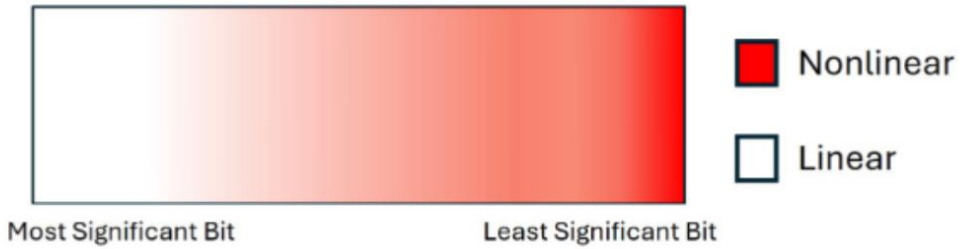
Outline

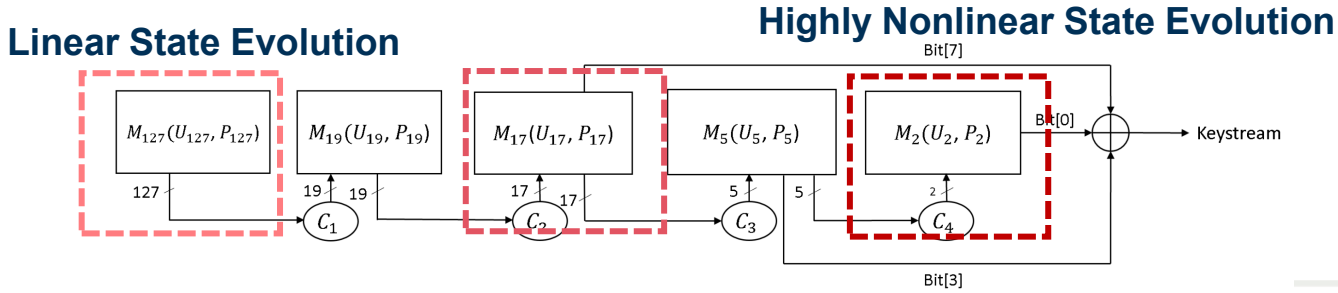
- Introduction
- Terminology and Notation
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Georgia Tech

Lightweight Cryptography using CMPRs: Gradient Nonlinearity

- **High-Level Takeaway:** For cryptographic purposes, it is helpful to view the state of a CMPR as a “nonlinearity gradient” to understand the fundamental structural differences between CMPRs and LFSRs/NLFSRs
  - CMPRs are based on a “feed-forward” chaining structure, whereas LFSRs/NLFSRs rely on a unified feedback function (linear for LFSRs, nonlinear for NLFSRs)
- Larger MPRs, which evolve under the influence of fewer chaining functions, update more linearly
- Smaller MPRs, which evolve under the aggregated influence of many chaining functions, update more nonlinearly
- This gradient view assumes the CMPR updates according only to the state update equations of the MPRs (e.g., no swapping)





Cryptanalytic Discoveries: Polynomial Representation of CMPRs

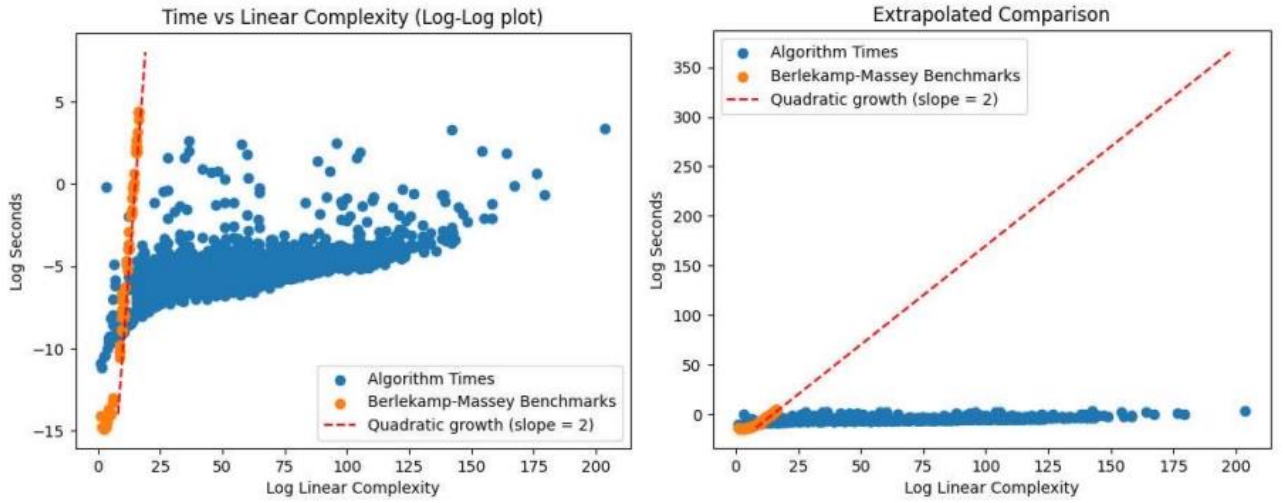
- Throughout the design process for the aforementioned CMPR-based cryptographic primitives, we have discovered several cryptanalytic properties specific to CMPRs
- From a cryptanalytic perspective, it is useful to model the different state bits of CMPRs as polynomials:
  - The **number of terms** in the polynomial representation of a state bit indicates the level of interdependence on the rest of the state, where more (independent) terms are desirable
  - The **algebraic degree** of the polynomial representation of a state bit indicates the level of nonlinearity for the evolution of that state bit, where higher degree is desirable
  - **Example:** $p(x) = x_1x_3x_7 + x_2x_0 + x_6$: $\deg\big(p(x)\big) = 3$, # of terms = 3
- The polynomial view of CMPRs is useful for analyzing susceptibility to **algebraic, linearization, and cube attacks**

Cryptanalytic Discoveries: Polynomial Representation of CMPRs

- To facilitate the generation of polynomials that accurately capture the I/O behavior of CMPRs, we leverage the Theorem 4 and the Root Expression Algorithm from [1]
  - Theorem 4 provides a methodology for obtaining the Z-transform of a feedback register that is chained into by a periodic sequence
  - The Root Expression Algorithm provides a scalable way for obtaining an upper bound on the linear complexity of sequences generated by a CMPR
    - Traditionally, the Berlekamp-Massey Algorithm [24] is used to gauge the linear complexity of a sequence by finding the smallest LFSR that can generate the sequence
    - The Berlekamp-Massey Algorithm has runtime $O(n^2)$, whereas our Root Expression Algorithm is polynomial with respect to the number of bits in the CMPR, and exponential with respect to the number of MPRs used [1], behaving sub-quadratically in practice

Cryptanalytic Discoveries: Polynomial Representation of CMPRs

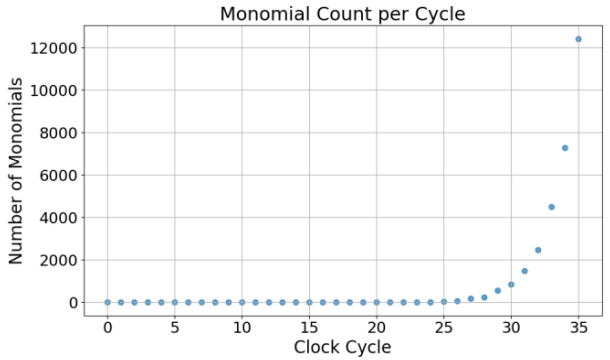
- Runtime and linear complexity comparison between the Berlekamp-Massey Algorithm and the CMPR Root Expression Algorithm applied to every possible CMPR configuration up to 300 bits [1]:



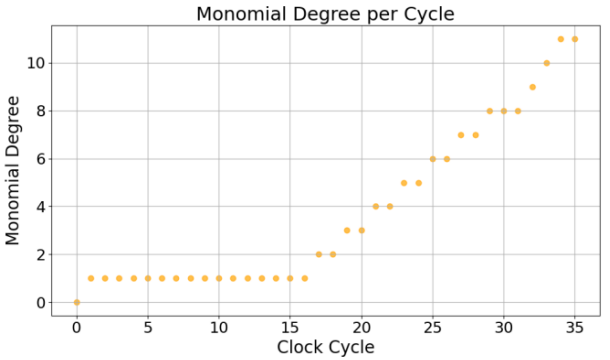
- **Key Idea:** The Root Expression Algorithm can be generalized to capture the specific monomials generated by a CMPR
- This generalization enables us to analyze the state (and state update) of CMPRs algebraically, which is a powerful cryptanalytic tool

Cryptanalytic Discoveries: Algebraic and Linearization Attacks

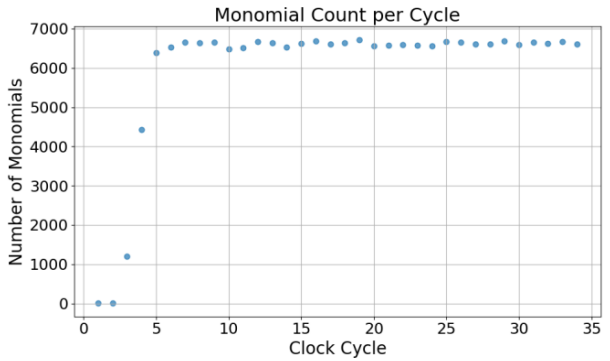
- Algebraic attacks [21] seek to analyze a system by constructing and solving a set of multivariate polynomial equations relating known variables to unknown variables
- Consider the polynomial representation of the LSB of a CMPR with respect to the (unknown) initial state variables
- We have determined that for CMPRs, the number of terms and degree of the polynomial representation grows rapidly with respect to clock cycles
    - The polynomial representation of an NLFSR, in comparison, grows slowly across many clock cycles
- This quick state mixing phenomenon observed in CMPRs complicates the application of algebraic and linearization/linear approximation-based attacks



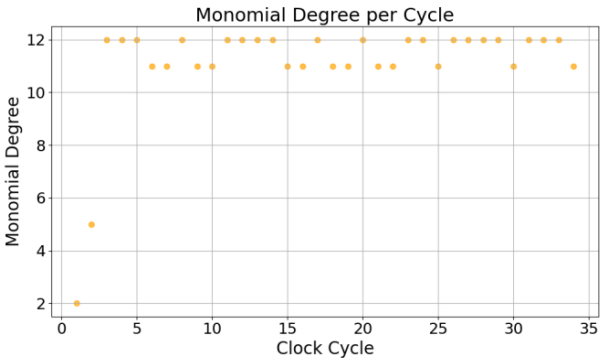
(a) Monomial Count



(b) Monomial Degree



(a) Monomial Count



(b) Monomial Degree

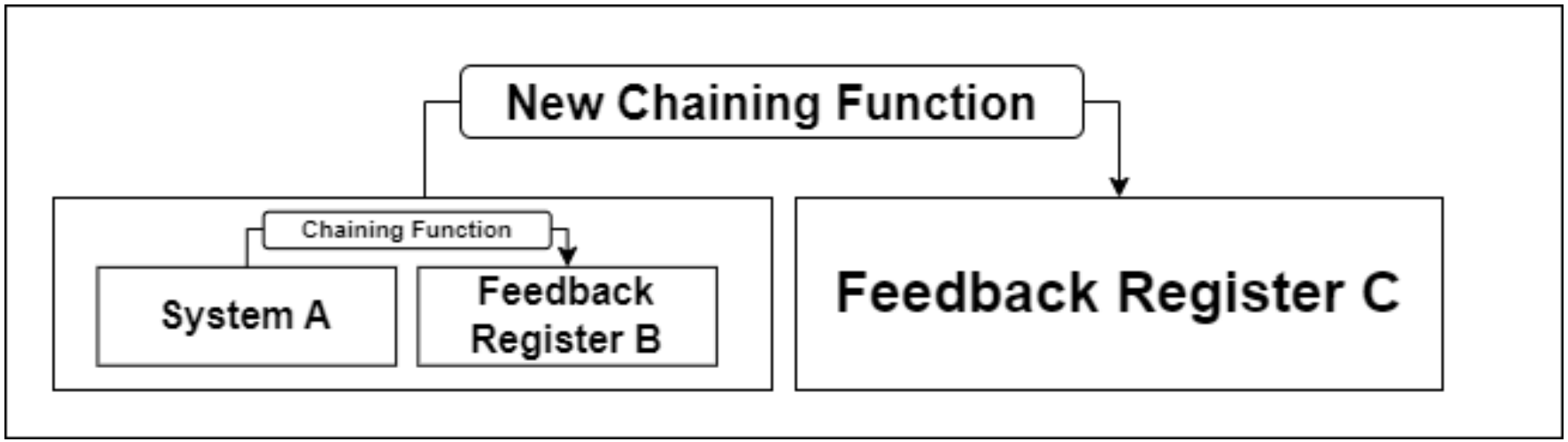
Georgia Tech

Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- Cube attacks [22] are a class of cryptanalytic attacks that model a system as a black-box polynomial and attempt to simplify the polynomial to the point where solving for the unknowns can be done efficiently
  - The black-box polynomial representation of a system models I/O relationships as an unknown polynomial whose degree is either known or can be easily guessed/bounded
- Cube attacks aim to represent evaluations of a black-box polynomial $p(x)$ in the form:
  - $p(x) = t_I p_{s(I)} + q(x)$
- and obtain a system of linear $p_{s(I)}$ in public and/or secret variables, to be solved
- **Example:** $p(x) = x_1x_2x_3 + x_1x_2x_4 + x_4x_0$ can be written as $p(x) = x_1x_2(x_3 + x_4) + x_4x_0$
  - Here, $t_I = x_1x_2$, and the linear $p_{s(I)} = x_3 + x_4$, and $q(x) = x_4x_0$

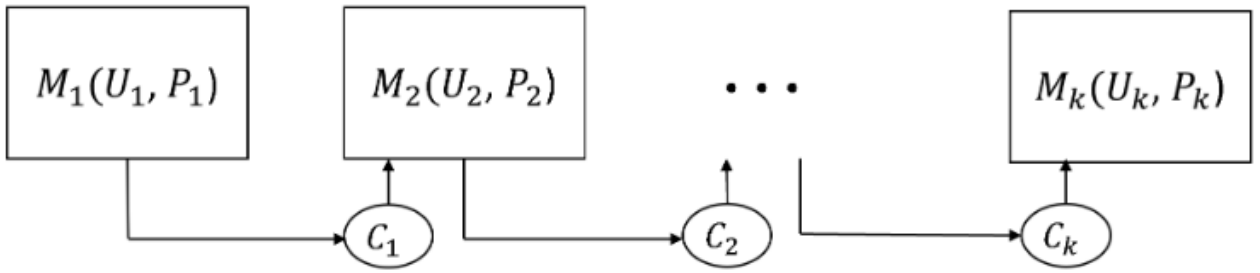
Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- In [1], we determined that CMPRs can be (not guaranteed to be!) vulnerable to cube attacks if the chaining functions in a CMPR construction are allowed to freely connect from a larger MPR to several smaller MPRs, especially if the chaining functions “skip” over some MPRs
  - Intuitively, this finding indicates that unrestricted chaining functions result in a low-degree black-box polynomial that can easily be simplified



Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- In [1], we determined that CMPRs can be (not guaranteed to be!) vulnerable to cube attacks if the chaining functions in a CMPR construction are allowed to freely connect from a larger MPR to several smaller MPRs, especially if the chaining functions “skip” over some MPRs
  - Intuitively, this finding indicates that unrestricted chaining functions result in a low-degree black-box polynomial that can easily be simplified
- Subsequently, we proposed the use of the following restricted chaining scheme to resist cube attacks:

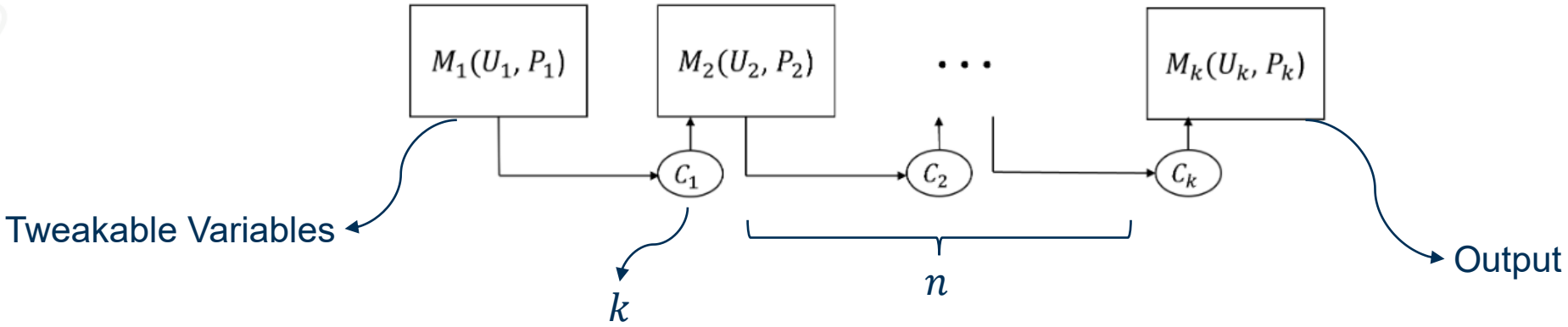


Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- Cube testers (or cube distinguishers) [22] are a class of attacks that aim to distinguish the output of a particular cryptosystem, represented as a black-box polynomial, from a truly random sequence
- In [1], we derived a closed-form expression for the attack complexity of cube distinguishers against CMPR-based PRNGs, with the following assumptions in mind:
  - Some portion of the CMPR state is initialized with “tweakable” (public) variables
  - The restricted chaining convention is followed
  - All chaining functions in the CMPR have the same upper bound for their degree
- Let k denote the (upper bound of the) degree of the chaining functions, and n denote the number of “fixed” (non-tweakable) MPRs between the last MPR containing tweakable variables and the MPR from which the output is extracted

Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- Let k denote the (upper bound of the) degree of the chaining functions, and n denote the number of "fixed" (non-tweakable) MPRs between the last MPR containing tweakable variables and the MPR from which the output is extracted



- Generally, if the degree of the output is d, the cube tester complexity is given by 2^{d+1} [23]
- For the scenario illustrated above, $\boldsymbol{d = k^{n+1}}$ [1] (neglecting corner cases)
- Thus, the cube tester complexity is $\boldsymbol{2^{k^{n+1}+1}}$ [1]
- **Example:** for $k = 4$, then $n = 2$ gives $> 2^{64}$ cube tester complexity

Cryptanalytic Discoveries: Cube Attacks and Distinguishers

- Our discoveries on cube attacks and distinguishers led us to provide the following design recommendations for cube attack/distinguisher resistance:
  - Ensure the existence of a **large gap in between MPRs initialized with cryptographic variables (key, IV, plaintext) and the MPR(s) from which cryptographic output is extracted**
    - Gap measured in terms of “fixed” MPRs, gauge resistance using the cube tester complexity equation
  - Alternatively, if the number of “fixed” MPRs cannot be increased, introduce **additional permutation operations** during an initialization phase, such as:
    - Swapping portions of the internal state
    - Carefully-selected feedback taps

Outline

- Introduction
- Terminology and Notation
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

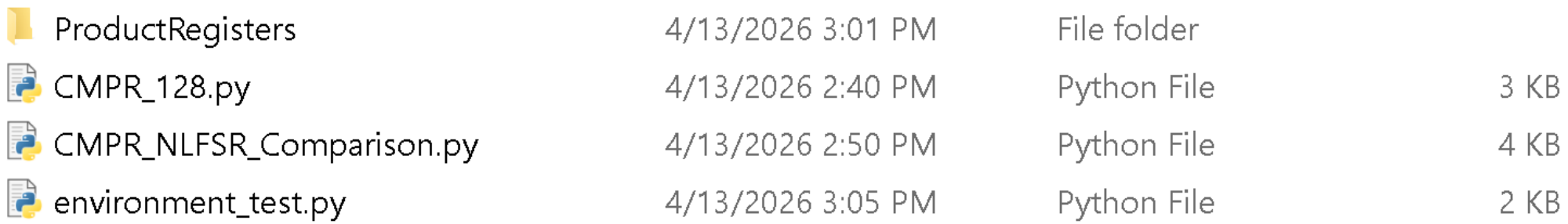
Georgia Tech

Interactive Demonstration: Toolchain Setup

- To prepare for the interactive portion of our tutorial, some toolchain setup is required
- The interactive portion can be performed on Windows, Linux, and macOS
- **Toolchain: Product Registers Python Library**
  - Navigate to https://github.com/gt-hwswcosec/ProductRegistersLibrary
  - Clone (download) the repository
  - Copy the “ProductRegisters” folder into an empty folder
  - Ensure you have Python 3.12 (or newer) installed on your system
  - Ensure you have pip 25 (or newer) installed on your system
  - Install the following pip packages, either using a virtual environment or globally (I have installed the packages globally):
    - memoization
    - numba
    - numpy
    - galois
    - python-sat
  - Navigate to https://github.com/gt-hwswcosec/HOST2026-CMPR-Tutorial/
  - Clone (download) the repository
  - Copy the Python scripts (CMPR_128.py, CMPR_NLFSR_Comparison.py, environment_test.py) to the same directory containing the “ProductRegisters” folder
    - (not into the “ProductRegisters” folder itself!)

Interactive Demonstration: Toolchain Setup

- Once setup, your environment should look like this:



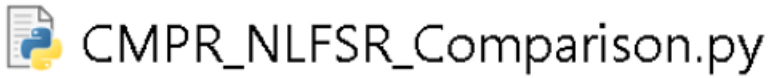
ProductRegisters	4/13/2026 3:01 PM	File folder	
CMPR_128.py	4/13/2026 2:40 PM	Python File	3 KB
CMPR_NLFSR_Comparison.py	4/13/2026 2:50 PM	Python File	4 KB
environment_test.py	4/13/2026 3:05 PM	Python File	2 KB

- To verify that the environment has been setup successfully, run environment_test.py:
  - Use a command line of your choice (I am using Windows PowerShell)
  - Use the Python command on your system corresponding to Python 3.12 or newer

```
PS C:\Users\arman\Documents\ECE\HOST2026_Tutorial> python .\environment_test.py
Environment test successful.
PS C:\Users\arman\Documents\ECE\HOST2026_Tutorial>
```

Interactive Demonstration: Script I

- The first portion of the interactive demonstration consists of running the "CMPR_NLFSR_Comparison.py" script



- This script instantiates the following feedback registers:
  - A 128-bit CMPR
  - A 128-bit NLFSR (sourced from Grain-128AEADv2 [6])
- Both registers are initialized to 0 in the LSB and 1's in all other bit positions
- Both registers are clocked 16 times and the state sequence for each register is printed
- Now, for each MPR object (lines 30-36), replace the third argument with `poly("x")`
- Replace line 45 with: `C128.generateChaining(template=prob_ANF_template(max_and=4, max_xor=4, p = 0.1))`
- **Key Takeaways:**
  - The state of the CMPR diffuses far more quickly than that of the NLFSR
  - The ability to tune chaining functions and update polynomials gives a designer the ability to balance the tradeoff between rapid state randomization and implementation cost

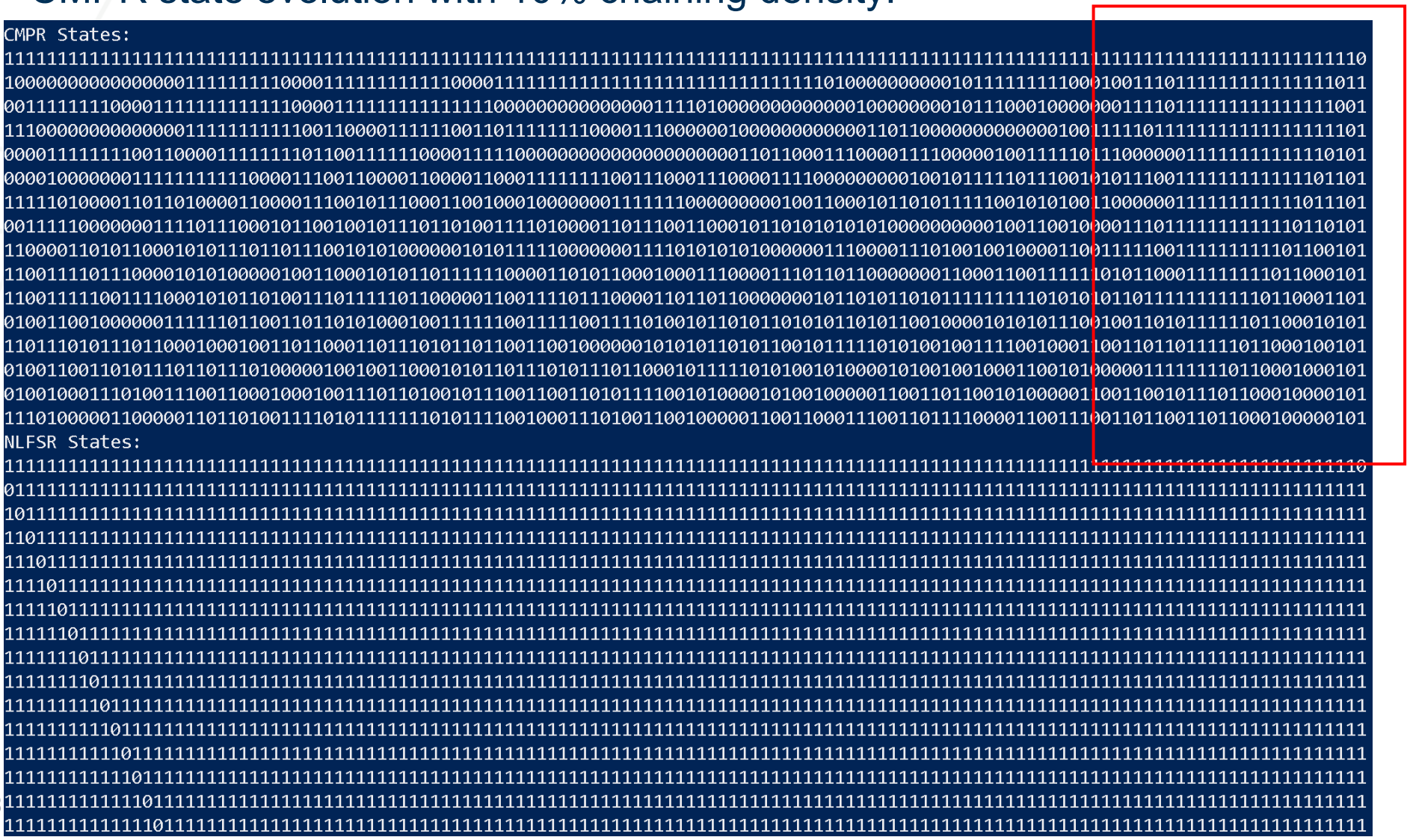
Interactive Demonstration: Script I

- CMPR state evolution with full-density chaining:

Georgia Tech

Interactive Demonstration: Script I

- CMPR state evolution with 10% chaining density:

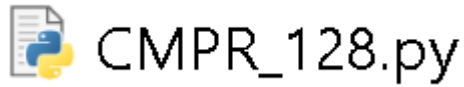


Slower diffusion in the LSBs due to fewer chaining functions

Georgia Tech

Interactive Demonstration: Script II

- The second portion of the interactive demonstration consists of running the “CMPR_128.py” script



  - This script instantiates a 128-bit CMPR constructed from 4 MPRs
  - Chaining functions are generated using AND gates with at most 4 inputs and XOR gates with at most 4 inputs
  - Once the CMPR has been instantiated:
    - Detailed logic expressions for each state bit are printed
    - VHDL for the CMPR is generated and stored in the file C128.vhd
  - **Key Takeaways:**
    - Each bit of the CMPR is a complicated function of other state bits, although the top-level MPR is updated linearly
    - The generated VHDL is modular: it can be simulated or synthesized on its own, or the VHDL logic expressions can be integrated into a larger design

Interactive Demonstration: Script II

- Example of the bitwise state update equation for the LSB:

```
Bit 0 updates according to:
(Main Function) = (
   XOR (
   |   XOR (
   |   |   AND (
   |   |   |   VAR(16)
   |   |   )
   |   )
   |   XOR (
   |   |   AND (
   |   |   |   VAR(33)
   |   |   |   VAR(28)
   |   |   |   VAR(34)
   |   |   |   VAR(32)
   |   |   )
   |   |   AND (
   |   |   |   VAR(31)
   |   |   |   VAR(24)
   |   |   |   VAR(26)
   |   |   |   VAR(27)
   |   |   )
   |   |   AND (
   |   |   |   VAR(26)
   |   |   |   VAR(30)
   |   |   |   VAR(33)
   |   |   |   VAR(34)
   |   |   )
   |   )
   )
)
```

$$a_0[t+1] = a_{16}[t] \oplus a_{33}[t]a_{28}[t]a_{34}[t]a_{32}[t] \oplus a_{31}[t]a_{24}[t]a_{26}[t]a_{27}[t] \oplus a_{26}[t]a_{30}[t]a_{33}[t]a_{34}[t]$$

- Results may vary for each different run of the script, each new run results in different chaining functions
  - To facilitate the persistent storage of chaining functions, the Product Registers Library supports JSON storage of CMPR objects
- Generated VHDL description of the 3 MSBs:

```
next_state(127) <= curr_state(111);
next_state(126) <= (curr_state(127) XOR curr_state(110));
next_state(125) <= (curr_state(126) XOR curr_state(109));
```

Outline

- Introduction
- Terminology and Notation
- Lightweight Cryptography using CMPRs
  - Stream Cipher Family Design
  - Authenticated Hybrid Stream Cipher Design
  - Hash Function Design
  - Cryptanalytic Discoveries
- Interactive Demonstration
- References

Georgia Tech

References

[1] D. Gordon, A. Allahverdi, S. Abrelat, A. Hemingway, A. Farooq, I. Smith, N. Arora, A. Chang, Y. Qiang, and V. Mooney, “Scalable nonlinear sequence generation using Composite Mersenne Product Registers,” *IACR Commun. Cryptol.*, vol. 1, no. 4, Jan. 2025, doi: 10.62056/a3tx11zn4.

[2] A. Allahverdi and V. Mooney, “A hardware-efficient AEAD stream Cipher based on a hybrid nonlinear feedback register structure,” *2025 IEEE International Conference on Cyber Security and Resilience (CSR),* Chania, Crete, Greece, 2025, pp. 1016-1023, doi: 10.1109/CSR64739.2025.11130096.

[3] I. T. L. Computer Security Division, “Lightweight Cryptography | CSRC,” *CSRC | NIST*, Jan. 03, 2017. https://csrc.nist.gov/projects/lightweight-cryptography

[4] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, “Submission to NIST,” 2019. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf

[5] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, “Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers.” Available: https://eprint.iacr.org/2014/386.pdf

[6] M. Hell *et al.*, “Grain-128AEADv2 -A lightweight AEAD stream cipher Cover sheet Corresponding submitter: Backup point of contact.” Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf

[7] E. Dubrova, “A list of maximum period NLFSRs,” *IACR Cryptology ePrint Archive*, 2012. [Online]. Available: https://eprint.iacr.org/2012/166.pdf

[8] E. Dubrova, “A scalable method for constructing Galois NLFSRs with period $2^n - 1$ using cross-join pairs,” *IEEE Transactions on Information Theory*, vol. 1, no. 59, pp. 703–709, 2013

[9] C. De Canniere, “Trivium: A stream cipher construction inspired byblock cipher design principles,” in *Lecture Notes in Computer Science*, vol. 4377, A. Biryukov, Ed. Berlin, Germany: Springer, 2006, pp. 171–186, doi: 10.1007/11836810 13.

[10] E. Dubrova and M. Hell, “Espresso: A stream cipher for 5G wireless communication systems,” *Cryptogr. Commun.*, vol. 9, no. 2, pp. 273–289, Dec. 2015, doi: 10.1007/s12095-015-0173-2

References

[11] Aumasson, JP., Henzen, L., Meier, W., Naya-Plasencia, M. (2010). QUARK: A Lightweight Hash. In: Mangard, S., Standaert, FX. (eds) Cryptographic Hardware and Embedded Systems, CHES 2010. CHES 2010. Lecture Notes in Computer Science, vol 6225. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-15031-9_1.

[12] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Public-Key Parameters. Boca Raton, FL: *CRC Press*, 1997, pp. 154–160. [Online]. Available: https://doi.org/10.1201/9780429466335.

[13] North Carolina State University. FreePDK45(TM), 2011. url: https://eda.ncsu.edu/freepdk/freepdk45/.

[14] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On the indifferentiability of the sponge construction," in *Advances in Cryptology – EUROCRYPT 2008*, N. Smart, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 181–197,ISBN: 978-3-540-78967-3.

[15] B. Mennink, R. Reyhanitabar, and D. Viz´ar, "Security of full-state keyed sponge and duplex: Applications to authenticated encryption," in *Advances in Cryptology– ASIACRYPT 2015*, T. Iwata and J. H. Cheon, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 465–489, ISBN: 978-3-662-48800-3.

[16] J.P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia, "ASCON v1.2: Submission to NIST," 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf

[17] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K.Yasuda, "PHOTON-Beetle authenticated encryption and hash family," 2021. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf

[18] C. Guo, T. Iwata, M. Khairallah, K. Minematsu, andT. Peyrin, "Romulus v1.3," 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf

[19] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, "Xoodyak, a lightweight cryptographic scheme," 2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf

[20] Terasic, Inc., "DE10-Standard," 2017. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English\&CategoryNo=165\&No=108

[21] Nicolas T. Courtois and Willi Meier. Algebraic Attacks on Stream Ciphers with Linear Feedback. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 345–359, Berlin, Heidelberg. Springer Berlin Heidelberg, 2003. doi: 10.1007/3-540-39200-9_21.

[22] Itai Dinur and Adi Shamir. Cube Attacks on Tweakable Black Box Polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 278–299, Berlin, Heidelberg. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-01001-9_16.

[23] Marco Cianfriglia, Elia Onofri, Silvia Onofri, and Marco Pedicini. Fourteen Years of Cube Attacks. Applicable Algebra in Engineering, Communication, and Computing, May 2023. doi: 10 . 1007 / s00200 - 023 - 00602 - w. url: https://doi.org/10.1007/s00200-023-00602-w.

[24] J. Massey, "Shift-register synthesis and BCH decoding," in IEEE Transactions on Information Theory, vol. 15, no. 1, pp. 122-127, January 1969, doi: 10.1109/TIT.1969.1054260.

[25] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak specifications," 2009. [Online]. Available: https://keccak.team/obsolete/Keccak-specifications-2.pdf