2022-03

ASIC BENCHMARKING FOR PROPOSED LIGHTWEIGHT CRYPTOGRAPHY STANDARD XOODYAK

Wakeland, Michael C.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ASIC BENCHMARKING FOR PROPOSED LIGHTWEIGHT CRYPTOGRAPHY STANDARD XOODYAK

by

Michael C. Wakeland

March 2022

Thesis Advisor:	Gaylord Henry
Co-Advisor:	Chad A. Bollmann

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2022	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE ASIC BENCHMARKING FOR PROPOSED LIGHTWEIGHT CRYPTOGRAPHY STANDARD XOODYAK	**5. FUNDING NUMBERS**
6. AUTHOR(S) Michael C. Wakeland	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

The U.S. National Institute of Standards and Technology (NIST) has initiated a process to standardize a "lightweight" cryptographic algorithm. Lightweight algorithms are designed for use in gate and performance-limited devices. This report compares an Application Specific Integrated Circuit (ASIC) implementation of the NIST Advanced Encryption Standard-128 (AES-128) and a competition finalist, Xoodyak. Implementations were written in SystemVerilog. Testing was performed using Vivado field programmable gate array simulations. Twenty six instances of AES and Xoodyak were built. These builds were optimized for throughput, clock frequency, and cell area, respectively. Size and performance benchmarks were obtained from builds using an 5nm and 16nm ASIC technology. Results indicate Xoodyak is capable of higher throughput than AES-128 while using a lower cell area.

14. SUBJECT TERMS lightweight, encryption, lightweight encryption, benchmarking, NIST, security, throughput, cell area, AES, Xoodyak, hardware, ASIC, SystemVerilog, hardware description	**15. NUMBER OF PAGES** 83
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

ASIC BENCHMARKING FOR PROPOSED LIGHTWEIGHT CRYPTOGRAPHY STANDARD XOODYAK

Michael C. Wakeland
Lieutenant, United States Navy
BS, University of Illinois at Urbana-Champaign, 2015

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
March 2022**

Approved by: Gaylord Henry
Advisor

Chad A. Bollmann
Co-Advisor

Douglas J. Fouts
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The U.S. National Institute of Standards and Technology (NIST) has initiated a process to standardize a “lightweight” cryptographic algorithm. Lightweight algorithms are designed for use in gate and performance-limited devices. This report compares an Application Specific Integrated Circuit (ASIC) implementation of the NIST Advanced Encryption Standard-128 (AES-128) and a competition finalist, Xoodyak. Implementations were written in SystemVerilog. Testing was performed using Vivado field programmable gate array simulations. Twenty six instances of AES and Xoodyak were built. These builds were optimized for throughput, clock frequency, and cell area, respectively. Size and performance benchmarks were obtained from builds using an 5nm and 16nm ASIC technology. Results indicate Xoodyak is capable of higher throughput than AES-128 while using a lower cell area.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AEAD	authenticated encryption with associated data
AES	Advanced Encryption Standard
AES-128	Advanced Encryption Standard, 128-bit security variant
ASIC	application specific integrated circuit
FSM	finite state machine
LWC	Lightweight Cryptography Competition
NIST	National Institute of Standards and Technology
RTL	register transfer level
S-box	substitution box

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The National Institute of Standards and Technology is holding a competition to create a lightweight cryptography standard. The scope of the competition contains both authenticated encryption with associated data, and hashing. Xoodyak is a cryptographic algorithm and finalist in the competition.

In this work, we wrote register transfer level code to reflect behavior for Xoodyak at a synthesizable hardware level. The complete module is operable by a user and capable of processing arbitrary length and numbered strings. We also wrote register transfer level code for the Advanced Encryption Standard, which is similarly operable. Centaur Technologies of Austin, Texas created 26 application specific integrated circuit builds based on this code, which were then compared on a throughput, cell area, and leakage power basis. The results show that Xoodyak builds can be optimized for size or throughput, and these tailored builds are either significantly smaller or faster than Advanced Encryption Standard across both 5nm and 16nm technology. Algorithm security was not evaluated within the scope of this research.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I wish to thank my advisors, G. Glenn Henry and Commander Chad Bollmann, for their mentorship over the last eighteen months. They showed great interest and vigor which inspired superior results. However, I think the most important thing is that I both learned something and thoroughly enjoyed the process. They greatly facilitated both goals. The experience was similarly valuable; not many had the chance to get a build of their graduate school projects.

I also wish to specifically recognize the efforts and resources of Jim Donahue and Centaur Technologies in making the builds. His builds furthered the success of this project and gave us the hard data we needed.

Lastly, I offer thanks for the fellowship of the following friends and colleagues who materially impacted my journey, both directly and indirectly.

LT Brandy Allain, USN

Ms. Marcea Ascencio, USAF

CAPT William Clive Blodgett, Jr., USN

CAPT Garrett Farman, USN

CAPT Dave Huljack, USN

Capitão-tenente Madjer & Juliana Martins, MDB

LT Karina Monroe, USN

Professor John Roth

CAPT Zoah Scheneman, USN

ENS Wyatt Smith, USN

LT Tim Stevenson, USN

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OVERVIEW

Encryption has been used to ensure the secrecy of information since antiquity. Two thousand years after the Caesar Cipher, the National Institute of Standards and Technology (NIST) adopted a subset of the Rijndael cipher as Advanced Encryption Standard (AES) [1]. AES is the most common and thoroughly vetted block cipher available today. Block ciphers ensure data confidentiality, but not integrity or authenticity.

TLS 1.3 and other modern security systems require Authenticated Encryption with Associated Data (AEAD), which provides integrity and authenticity in addition to confidentiality [2]. The use of AES with AEAD requires more than the AES block cipher alone[1] [3]. These additional requirements can limit throughput and add computational burden to resource-constrained devices. The NIST Lightweight Cryptography Competition (LWC) seeks to alleviate these concerns by creating a standard for both AEAD and hashing [4].

Daemen et al. created Rijndael as well as Xoodyak, which is one finalist in the NIST competition [5]. Xoodyak is a family of mathematical functions that are synthesizable in hardware. Xoodyak can be used for both AEAD and hashing. The principal characteristic in Xoodyak is a 384-bit *State*. The *State* undergoes minor modification at the start of every function call and is then distributed by a permute function which performs various Boolean XORs and round key additions in 12 iterations. The output is then combined with an input text to "absorb" a string or generate an output text. This process repeats, perhaps multiple times, depending on the length of the input string, until all text is processed. ASIC benchmarking can determine the performance relationship between Xoodyak and AES, which is the primary goal of this work.

[1] NIST recommends the use of Galois Counter Mode to mode to perform AEAD with an AES core, as described in ref. [3], NIST Special Publication 800–38D Ch 7: GCM Specification

B. MOTIVATION

The Fleet has a need for lightweight, scalable encryption to support time-critical and Internet of Things applications. Higher throughput is one of the goals of the LWC, which translates to smaller encryption delays in operational use. If selected as the lightweight cryptographic standard, Xoodyak will be positioned to provide AEAD with low latency to fill these requirements. Quantifying the relative gain over the existing standard, AES, is a critical benchmark for assessing algorithm performance.

The hardware builds which we created will also provide the NIST Lightweight Cryptography Competition with ASIC benchmarking data. This data will better inform NIST decision makers about real performance marks for Xoodyak in an isolated standard-to-proposed-standard comparison in relation to AES-128.

C. SCOPE

This research evaluated the performance and size of the AES-128[2] standard in comparison to the Xoodyak cryptography suite.[3]

This comparison was selected because AES is the established technical standard, and Xoodyak is a proposed technical standard for lightweight cryptography. Consequently, the direct comparison is between an established and proposed standard.

ASIC benchmarks were performed by Centaur Technologies of Austin, Texas. Our research was concerned with implementing cores and measuring parameters to assess performance. These parameters were cell area, cell count, and clock frequency. The research did not evaluate any dynamic power consumption or security claims.

[2] In the context of this document, "AES," "AES core," or "AES standard" refers to statements about the entire Advanced Encryption Standard regardless of key size (128, 192, or 256 bits). Every use of AES refers to the block cipher only, not AES as a part of a larger AEAD scheme such as that described in the NIST Recommendation [3]. "AES-128" specifically refers to an AES standard algorithm using a 128-bit key size, or a build that only supports 128-bit true keys.

[3] As described in "Xoodyak, a Lightweight Encryption Scheme." - Algorithm 2, page 9 [5]. The suite includes every possible combination of functions with arbitrary input lengths and arbitrary order, so long as the order is algorithmically feasible.

D. THESIS ORGANIZATION

Chapter II describes the control implementation of the AES core, which includes the *KeyExpansion*, *Encrypt*, and *Decrypt* modules. Chapter III describes the Xoodyak algorithm in theory and hardware architecture. Chapter IV describes full benchmarking results and comparisons. Chapter V is the conclusion. Appendix A contains a link to the author's GitHub repository containing the full register transfer level (RTL) description for both AES and Xoodyak. Appendix B is a reproduction of the Xoodoo *Permute* function as described in [3]. Appendix C is an example series of function calls, evaluated using the Xoodyak hardware instance. Appendix D provides the technical build methodology employed by Centaur Technologies in creating the hardware instances. Appendix E contains the full list of AES-128 and Xoodyak builds and benchmarking data. Appendix F provides selected die plot images of various hardware builds.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. OVERVIEW

The purpose of this research was to evaluate Xoodyak performance in relation to AES. A direct evaluation required RTL instantiations of both algorithms and subsequent builds for benchmarking. A "build" is the collection of transistors and wires which perform the described function, demonstrating the size and performance on any resulting silicon chip. Only minor steps remain before a build can be released into manufacturing. This chapter concerns the AES instance, from both a structural overview and user operation perspective.

The AES algorithm supports 128, 192, and 256-bit key lengths, but the 196 and 256-bit key lengths require more hardware to implement. They also require more rounds, or iterations of the algorithm, to complete an encryption cycle, which increases cell area and decreases throughput. Xoodyak only claims 128 bits of security. Our AES implementation therefore only supports 128-bit key lengths to obtain a more equivalent perspective.

The AES structure is designed to accomplish one AES round every clock, as described in the AES Standard. We also made several design decisions, particularly regarding substitution-boxes (S-boxes) which are discussed in more detail in Section II.C.

The separate *KeyExpansion, Encrypt*, and *Decrypt* modules are wired together in a manner that supports continuous operation. The full AES module is fully operational; users can expand keys, then encrypt or decrypt an arbitrary length string.

Lastly, the completed AES instance was verified using a combination of the sample test vectors in the AES Standard and hand verification using MATLAB.

B. AES IMPLEMENTATION

NIST FIPS - 197 forms the basis of AES [1]. AES, including AES-128, is publicly understood. Our discussion regarding AES is therefore limited to general approach, functionality, and notable design decisions.

1. Round Based

Our implementation of AES sought to match the technical requirements of [1] in a functional, ASIC-instantiated RTL model. The implemented datapath closely resembles the steps in [1] for all modules. Each module uses a round-based approach, where one clock completes one round. For example, one clock in the *Encrypt* module accomplishes one *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey* transformation. Ten clocks are required to accomplish the entire *Cipher* (*Encrypt*) operation. *Inverse Cipher* (*Decrypt*) and *KeyExpansion* require ten and 11 clocks respectively under the same paradigm.

2. Functionality

Our AES-128 implementation includes hardware to implement the *KeyExpansion*, *Encrypt*, and *Decrypt* functions. The modules are interlinked for operational use. Users can call various functions based on a 2-bit opcode. The appropriate data must be provided synchronously with the function call. The opcodes are listed in Table 1.

Table 1. AES-128 Function Call Decodes

Functions [1:0]	
Opcode	AES
0	Idle
1	Encrypt
2	Decrypt
3	Keys

Ingesting a valid opcode creates a text output for *Encrypt* or *Decrypt* calls. Function durations are counter-based and determinate. Opcode 3 calls for *KeyExpansion* create no output text because the expanded keys are held in the module for future use. However, an output text decoder informs the user when *KeyExpansion* is complete. The return decode is provided synchronously with the output text for *Encrypt* and *Decrypt* calls. Table 2 is a full readout of output function decodes. Throughput and cell area results are detailed in Chapter IV.

Table 2. AES-128 Function Return Decodes

Returns [1:0]	
Decode	AES
0	Invalid
1	Ciphertext
2	Plaintext
3	Keys

C. HARDWARE DESIGN DECISIONS

Several design decisions had to be made when designing the AES hardware instance. For example, accomplishing one AES round per clock was a conscious choice. Two deliberate choices involved how to handle the forward and inverse S-box, and whether to reverse certain commutable steps in the *Decrypt* module. The most significant design choice was to hold expanded keys in a register, rather than generate keys "on the fly."

1. S-box Instance

The *Encrypt* module contains a forward Rijndael S-box instance. The S-box is shared with the *KeyExpansion* module for use in key generation. The Inverse S-box is separately instanced in the *Decrypt* module, rather than being computed as a derivative of the forward S-box.

2. Default Cipher Configuration

Certain AES steps are commutable [1]. This implementation performed the standard function flows with no use of "equivalencies." The *Encrypt/Decrypt* datapath flows are summarized in Table 3.

Table 3. AES Datapath Flow Order

	AES Flow	
Step	Encrypt	Decrypt
1	SubBytes	InvShiftRows
2	ShiftRows	InvSubBytes
3	MixColumns	AddRoundKey
4	RoundKey	InvMixColumns

Table reflects both the default datapath per the AES Standard, and the implementation in the *Encrypt* and *Decrypt* Modules. Adapted from [1].

3. Vector Key Generation

Keys are provided in a complete set to the *Encrypt* and *Decrypt* modules. Calls to *Encrypt* or *Decrypt* use the stored keys for computation. One key is generated on every clock. The expansion is complete when all 11 keys are generated. The keys are held in registers, comprising 11 128-bit registers in total. These stored keys comprise a large part of the *KeyExpansion* hardware. The keys are not "flushed" after an encryption cycle. This allows multiple cipher operations to use the same key vector. Figure 1 visualizes both how key vectors are shared with both *Encrypt* and *Decrypt*, and shows the high-level structure of the AES-128 module.

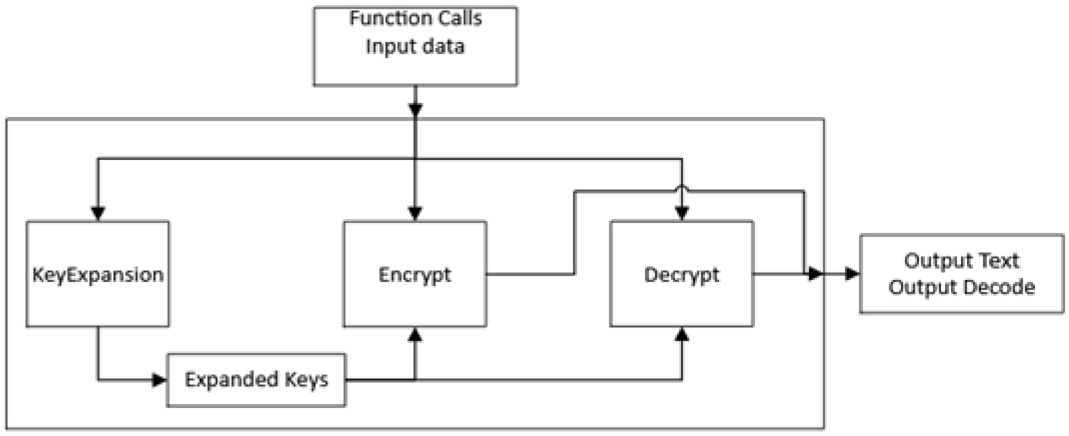


Figure 1. Top Level AES-128 Process Diagram

D. AES BEHAVIORAL VERIFICATION

Our AES-128 instance supports three function calls and idle. Vivado is an RTL design suite by Xilinx for analysis and synthesis of RTL logic [6]. Each function was individually verified through Vivado using MATLAB, [1], and [7]. The MATLAB script in [7] was assumed to be correct after our detailed visual inspection and verification of intermediate and final outputs from the sample test vectors in [1]. Our individual vector results were verified by hand using a combination of the MATLAB code in [7] and Vivado waveform simulation. The waveform window was examined for correct output text, opcode/decode/operation congruence, and timing. Figure 2 visualizes this process.

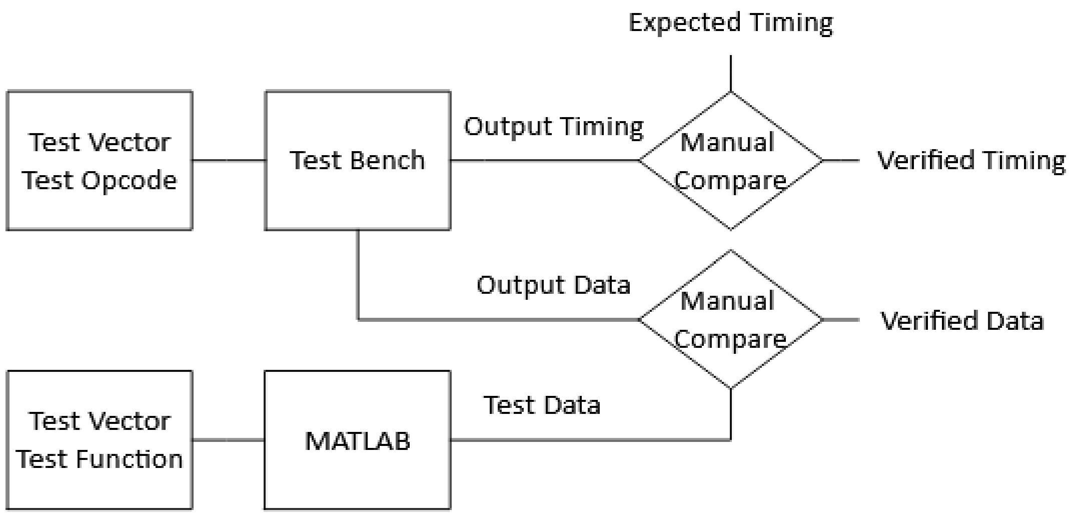


Figure 2. Function Verification Process Flow Diagram

After individual functions were validated, all possible sequences of function calls were performed in Vivado with MATLAB verification. The data used was the default test vector in the AES Standard [1]. The nine distinct function transitions were performed to directly test for errors in control logic. Table 4 describes every possible function transition. Idle transitions are not included because they are no-ops. The output pins were then examined for correctness across the entire function sequence.

Table 4. Validated Function Transitions

Function transitions	
Call N	Call N+1
KeyExpansion	KeyExpansion
KeyExpansion	Encrypt
KeyExpansion	Decrypt
Encrypt	KeyExpansion
Encrypt	Encrypt
Encrypt	Decrypt
Decrypt	KeyExpansion
Decrypt	Encrypt
Decrypt	Decrypt

E. SUMMARY

This chapter discussed a stable control to compare with Xoodyak, the proposed lightweight cryptography standard. Our hardware instance of AES-128 serves as that control basis. This operational AES-128 module forms a basis on which to compare to an operational Xoodyak module. The algorithmic definition and hardware construction of Xoodyak is the topic of the next chapter.

III. METHODS

A. OVERVIEW

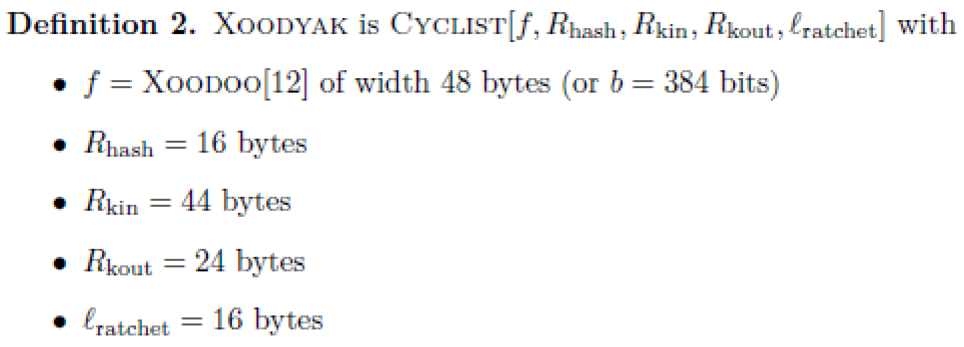
The purpose of this research was to benchmark the Xoodyak suite against the AES-128 block cipher. The principal component of this work was the development of a Xoodyak hardware instance. Creating the Xoodyak instance required two main efforts. The first part was understanding the Xoodyak algorithm and making design decisions on supportability. This required a thorough investigation of the formal algorithm definitions which make up Xoodyak which are described in Section B.

The second part was writing the RTL code that supported those requirements. Section E describes the implementation from a structural and operational perspective using RTL logic in synthesizable SystemVerilog. Appendix A contains links to the full RTL description. These new implementations were built using the same tools and circuit modules as our AES-128 instance.

The Xoodyak module is fully operational, just like our AES-128 instance. This includes the capability to handle arbitrary length strings across all AEAD and hashing functions using two inputs, text, and opcode. Verification was conducted by hand examination and C code. An example series of function calls with associated data is available in Appendix C.

B. THE XOODYAK ALGORITHM

The Xoodyak algorithm is an instance of the *Cyclist* algorithm with specific parameters [5]. *Cyclist* is made up of a series of mathematical function calls. The algorithmic instantiation of Xoodyak is given in Figure 3.



Definition 2. Xoodyak is Cyclist$[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$ with

- $f =$ Xoodoo[12] of width 48 bytes (or $b = 384$ bits)
- $R_{\text{hash}} = 16$ bytes
- $R_{\text{kin}} = 44$ bytes
- $R_{\text{kout}} = 24$ bytes
- $\ell_{\text{ratchet}} = 16$ bytes

Figure 3. Definition of Xoodyak as *Cyclist* with Parameters. Source: [5].

Xoodyak, being a derivative of the *Cyclist* algorithm, operates through what we termed "level 1" function calls.[4] We have parsed these functions according to levels for clarity, but this is not a distinction that exists in the NIST submission. All functions, and their precise relationships, are given in Algorithm 2 of [5]. Level 1 function calls are similar to *KeyExpansion*, *Encrypt*, or *Decrypt* function calls in AES. Level 1 functions take user inputs and create outputs. Lower-level functions are described in Algorithm 3 of [5] as "internal interfaces."

The higher-level function calls heavily overlap with lower-level calls. Two examples of level 1 functions include *Encrypt* and *Decrypt*, which both call the *Crypt* function on level 2. Every level 1 function call calls *Xoodoo*, the *Permute* core of Xoodyak, if the phase is "DOWN" at the start of the call.[5] This also sets the Xoodyak phase to "UP." Table 5 shows the relative relationship between all the various nested functions.

[4] Function calls in this section specifically refer to *algorithm* function calls, not hardware functions called by an opcode and supplied by a user.

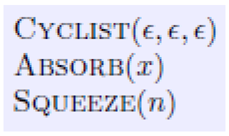
[5] *Xoodoo* and *Permute* are used interchangeably in this document. Likewise, permutation refers to the results of *Xoodoo*, and the verb "to permute" a string means to provide a *State* input to *Xoodoo* and collect the result.

Table 5. Organization of Xoodyak Function Calls. Adapted from [5].

Top Level Instantiation			
Cyclist (keyed)		Cyclist (Hash)	
Level 1			
AbsorbKey	Decrypt	Squeeze	Ratchet
Absorb	Encrypt	SqueezeKey	-
Level 2			
AbsorbAny	Crypt	SqueezeAny	-
Level 3			
Split	Down	Up	-
Core			
f (Xoodoo)	-	-	-

The Daemen et al. formal Xoodyak submission is described by a specific series of function calls. However, users can call functions in arbitrary order. For example, while {*Cyclist*, *Crypt*, *Crypt*, *Absorb*, *Squeeze*, *Absorb*} is not a series of function calls in the AEAD or hash submission, it is still algorithmically possible. The formal AEAD and hash function calls for the submission are described, along with their parameters, in Figures 4 and 5.

When used in hash mode as described in Figure 3, the Xoodyak algorithm can produce any *n*-byte hash value, based on the absorption of any arbitrary length associated data input *x* [5]. The *Cyclist* function generates a string of all zeros as the arguments are null. The *Absorb* function call described absorbs an arbitrary length string *x* into the *State*. The *Squeeze* function call generates an *n*-byte length hash, depending on the *State*.

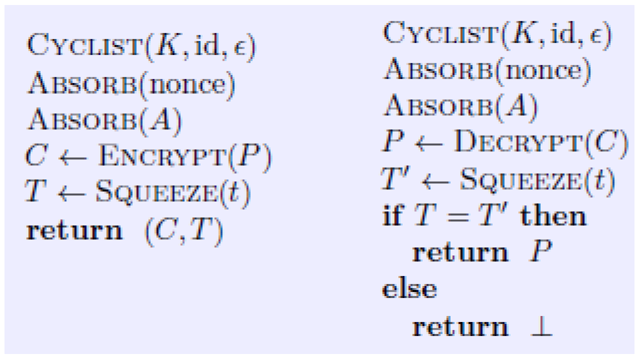


```
CYCLIST($\epsilon, \epsilon, \epsilon$)
ABSORB($x$)
SQUEEZE($n$)
```

Figure 4. Use of Xoodyak Functions for Hashing. Source: [5].

Figure 4 describes the use of Xoodyak in AEAD. The fundamentals of the process are the same, supplying arguments represented by a variable name. AEAD requires a key

K, which is null in hash mode. Several inputs are parameterized. The parameters are discussed in greater detail in the implementation, Section E.1.



```
$\text{Cyclist}(K, \text{id}, \epsilon)$
$\text{Absorb}(\text{nonce})$
$\text{Absorb}(A)$
$C \leftarrow \text{Encrypt}(P)$
$T \leftarrow \text{Squeeze}(t)$
return $(C, T)$
```

```
$\text{Cyclist}(K, \text{id}, \epsilon)$
$\text{Absorb}(\text{nonce})$
$\text{Absorb}(A)$
$P \leftarrow \text{Decrypt}(C)$
$T' \leftarrow \text{Squeeze}(t)$
if $T = T'$ then
  return $P$
else
  return $\perp$
```

Figure 5. Use of Xoodyak Functions for AEAD. Adapted from [5].

C. PRIMITIVES

A common thread unites Xoodyak operations: the relationship between two objects and a permutation. The first object is the *State*, which is a vector that changes value depending on every function call since instantiation. The second are color bytes, which provide domain separation at the start of each function call. In any function, the *State* undergoes a color byte modification, is permuted, and then has another color byte modification.[6]

1. *State* (as a concept)

The *State* is the fundamental object described in Xoodyak. The *State* has three properties: value, length, and phase. In Xoodyak, the *State* is always 384 bits long and the value is determined by process history. The process history is the sequence of function calls, in order, with arguments, up to the present call. The phase of *State* is either "UP" or "DOWN." The phase can be flipped from "UP" to "DOWN" by the *Down* internal function, or from "DOWN" to "UP" by the *Up* internal function. Applying the *Up* function

[6] The color bytes are only active the first "pass" through a function. For strings of greater length than the maximum operation length (given by parameters in Figure 3, Table 6) multiple passes are required.

to *State* whose phase is "UP" creates a no-op. Applying the *Down* function to *State* whose phase is "DOWN" is not algorithmically possible.

2. Color Bytes

An *Up* color byte, C_U, is applied via bitwise XOR to the least significant byte of the *State* depending on the function called. For operations with multiple calls to *Up* and *Down*, the C_U is only applied before the first call to *Up*.

3. Core f

The *f* function is an instance of the *Xoodoo/Permute* function summarized by Algorithm 1 [5]. It manipulates the *State* based on a set operation and pre-defined round constants. The *State* manipulation performed by *Xoodoo* forms the core of the Xoodyak algorithm. For Xoodyak, 12 rounds are required to complete the *f* function. Algorithm 1 is reproduced fully in Appendix B.

D. DESCRIBING A FUNCTION CALL

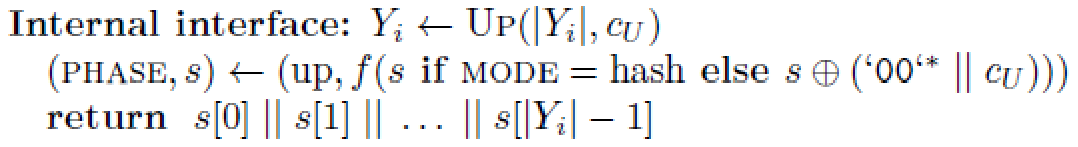
Level 1 function calls perform a modification of the *State*, then permute the *State* through *Up* and then D*own*. Function calls with large-sized arguments endure this process multiple times. The precise nature of whether an *Up* function is used on a call, or what modifications are performed before or after it, are specific to the function, the length of the function call, and the phase of the *State* when the call is processed. Each level of function call must be described in detail. The discussion must begin at the lowest level with *Permute,* and end at the highest level with *Cyclist*, because each function can call all lower-level functions.

1. Level 3 Function Calls

a. UP

Up calls permute, change the phase to "UP," and updates the value of the *State* with the post-permute value. Setting the phase to "UP" prevents a subsequent *Up* call until a *Down* function processes. *Up* accepts |Yi| and the *Up* color byte C_U as inputs, where |Yi| is the size of the desired return value in bytes. In practice this is a parameter based on the function call.

CU is not applied to the *State* in hash mode, but it is applied to the least significant byte of the *State* in keyed mode. The returned value from *Up* is the most significant |Yi| bytes of the state, after the permute. Figure 6 gives the formal definition for *Up*.



```
Internal interface: $Y_i \leftarrow \mathrm{UP}(|Y_i|, c_U)$
  $(\mathrm{PHASE}, s) \leftarrow (\mathrm{up}, f(s \text{ if } \mathrm{MODE} = \mathrm{hash} \text{ else } s \oplus (\text{‘00‘}^* \,||\, c_U)))$
  return $s[0] \,||\, s[1] \,||\, \ldots \,||\, s[|Y_i| - 1]$
```

Figure 6. *Up* Function Definition. Source: [5].

b. *DOWN*

The *Down* function performs post-permute processing. At a high level, the *Down* function creates a vector, then XORs that vector with the *State*. The output of this XOR is the new *State*.

The entire post-permute processing has four parts concatenated together in a specific order. In Xoodyak, this concatenated vector is 384 bits long and articulates which bits to flip in the *State*. The first is the input data, X_i, is of variable length and can be zero. The second is 0x01. The third is an extended 0x00 which spans the distance from the end of the second part to the final byte. The last part is a *Down* color byte (*CD*), which also contains a logical AND with 0x01 in hash mode.[7] *CD* varies depending on the function call. Figure 7 describes this process in formal terms, where both the phase and state (s) are assigned.

[7] Contextually, the impact of the logical AND 0x01 in hash mode only means that *CD* is zero for all hash mode functions, except for the first pass through a call to *Absorb*. Recall that the only valid functions in hash mode are *Cyclist, Absorb,* and *Squeeze.*

Internal interface: $\textsc{Down}(X_i, c_D)$

$(\textsc{phase}, s) \leftarrow (\text{down}, s \oplus (X_i \,||\, \text{‘01‘} \,||\, \text{‘00‘}^* \,||\, c_D \ \&\ \text{‘01‘ } \textbf{if } \textsc{mode} = \text{hash } \textbf{else } c_D))$

Figure 7. *Down* Function Definition. Source: [5].

c. *SPLIT*

The *Split* function breaks the input string *X* into *n* sized pieces. Argument *n* is the maximum length that can be operated on at a single time and is variable depending on the function called. An *n*-sized piece of the string is processed for every sequential alteration through *Down* and *Up* until the entire input string *X* is exhausted.

2. Level 2 Function Calls

The level 2 functions are called by the level 1 functions. Level 2 function calls perform a modification of the *State*, then permute the *State* through *Up*, then perform another modification of the state through *Down*. The process is repeated depending on the input strings. The difference between the level 2 functions is the nature of the arguments supplied to the modifications in pre- or post-permute processing, and the length of the parameters supplied to those modifications.

a. *AbsorbAny*

AbsorbAny adds an input string to the state. Like every level 2 function, arguments are supplied by the level 1 functions. Until the input string is exhausted, *AbsorbAny* will break the first input into |*X*/r| sized chunks through the Split function, then apply the *Up* (which includes the permute) and *Down* functions to the state for every chunk. If the phase is “UP” at the start of the function call, the *Up* function is not called and the function proceeds directly to *Down*.

The C_D is zero for every run through the *Up* and *Down* function beyond the first one. Regardless of input string size, *AbsorbAny* always terminates with the state in the “DOWN” phase. Figure 8 gives the formal definition for *AbsorbAny*.

```
Internal interface: $\text{AbsorbAny}(X, r, c_D)$
  for all blocks $X_i$ in $\text{Split}(X, r)$ do
    if PHASE $\neq$ up then $\text{Up}(0, \text{‘00‘})$
    $\text{Down}(X_i, c_D$ if first block else ‘00‘)
```

Figure 8. *AbsorbAny* Function Definition. Source: [5].

b. *Crypt*

Crypt performs either the encrypt or decrypt operation. The calling level 1 function supplies text *I* and a Boolean value indicating whether the text is to be enciphered into the *State* (0 for encrypt), or whether the text is ciphertext to replace in the *State* (1 for decrypt).

The text is broken into ($I/|R_{kout}|$) chunks, rounded up, through *Split* as described before. Then for each chunk, *Up* is performed on the state.

For *Encrypt* operations, the argument supplied to the *Down* function is the input text, parted into chunks. For *Decrypt* operations, the input to the *Down* function is input text XORed with the most significant $|\mathrm{R}_{\mathrm{kout}}|$ bytes of the *State*. Behaviorally, this means that the first $|\mathrm{R}_{\mathrm{kout}}|$ bytes of the state are replaced with the input chunk in *Decrypt* operations. Figure 9 gives the formal definition for *Crypt*.

```
Internal interface: $O \leftarrow \text{Crypt}(I, \text{DECRYPT})$
  for all blocks $I_i$ in $\text{Split}(I, R_{\text{kout}})$ do
    $O_i \leftarrow I_i \oplus \text{Up}(|I_i|, \text{‘80‘ (crypt) if first block else ‘00‘})$
    $P_i \leftarrow O_i$ if DECRYPT else $I_i$
    $\text{Down}(P_i, \text{‘00‘})$
  return $||_i\, O_i$
```

Figure 9. *Crypt* Function Definition. Source: [5].

c. *Squeeze*

Squeeze produces an *l*-byte hash of the state. Like all level 2 functions, the supplied arguments are defined by the level 1 functions. The first *Up* call permutes the *State*, and the most significant $|R_{squeeze}|$ bytes are held as the hash output. If *l* is smaller than $R_{squeeze}$ then *L* is used instead.

If the size of the hash is less than the desired value *l,* the *Down* function applies, which as defined in *Squeeze* only flips the eighth most significant bit in the state due to the specific arguments supplied.

The *Up* function is then called again, and once more the first $|R_{squeeze}|$ bytes are appended to the hash. The cycle of *Down* followed by *Up* continues until the desired hash length is obtained.

In every case the phase ends with "UP" when *Squeeze* is called. Figure 10 gives the formal definition for *SqueezeAny*.

```
Internal interface: $Y \leftarrow \text{SqueezeAny}(\ell, c_U)$
  $Y \leftarrow \text{Up}(\min(\ell, R_{\text{squeeze}}), c_U)$
  while $|Y| < \ell$ do
    $\text{Down}(\epsilon, \text{`00`})$
    $Y \leftarrow Y \,||\, \text{Up}(\min(\ell - |Y|, R_{\text{squeeze}}), \text{`00`})$
  return $Y$
```

Figure 10. *SqueezeAny* Function Definition. Source: [5].

3. Level 1 Function Calls

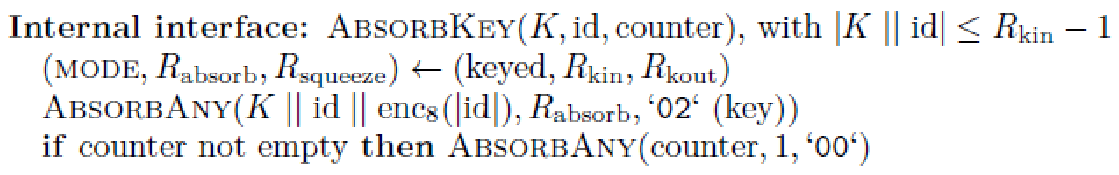
Level 1 function calls supply arguments to and invoke the level 2 function calls. As such, their descriptions are curt.

a. AbsorbKey

AbsorbKey is called to bring the secret key into the state. *AbsorbKey* can only be called as part of the initial state instantiation. *AbsorbKey* initializes the state in keyed mode, and applies the defined parameters, R_{kin} and R_{kout}, to R_{absorb} and $R_{squeeze}$ (the *Absorb* and *Squeeze* rate every permutation).

The items are supplied to *AbsorbAny* as a single argument. They take the form (*K* || id || enc8(|id|)), where K is the key, || is the concatenation operator, and enc8(|id|) is the modulo 256 value of the size of the optional identifier, id. Xoodyak default parameters presume id to be null .

The second argument is R_{absorb} is always 16 bytes or Xoodyak AEAD, and 0x02 is the color byte for *AbsorbKey*. The counter field is used as another way to absorb a nonce as described in [5]. Figure 11 gives the formal definition for *AbsorbKey*.

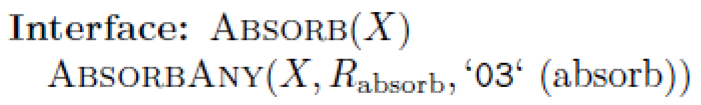


```
Internal interface: AbsorbKey($K$, id, counter), with $|K \,||\, \text{id}| \leq R_{\text{kin}} - 1$
    (mode, $R_{\text{absorb}}$, $R_{\text{squeeze}}$) $\leftarrow$ (keyed, $R_{\text{kin}}$, $R_{\text{kout}}$)
    AbsorbAny($K \,||\, \text{id} \,||\, \text{enc}_8(|\text{id}|)$, $R_{\text{absorb}}$, '02' (key))
    if counter not empty then AbsorbAny(counter, 1, '00')
```

Figure 11. *AbsorbKey* Function Definition. Source: [5].

b. *Absorb*

Absorb is the level 1 function the user invokes to absorb a string *X* into the state. Its only function is to immediately call the level 2 function *AbsorbAny*, where the arguments are the input string *X*, the parameter R_{absorb} as defined in the *Cyclist* call to *AbsorbKey*, and '03' for the color byte. Figure 12 gives the formal definition for *Absorb*.



```
Interface: Absorb($X$)
    AbsorbAny($X$, $R_{\text{absorb}}$, '03' (absorb))
```

Figure 12. *Absorb* Function Definition. Source: [5].

c. *Encrypt*

The only argument for *Encrypt* is the text to be encrypted. *Encrypt* can only be called in keyed mode. It passes the text to *Crypt* as well as a Boolean 0 to indicate encryption vice decryption. Figure 13 gives the formal definition for *Encrypt*.

```
**Interface:** $C \leftarrow \textsc{Encrypt}(P)$, with $\textsc{mode} = \text{keyed}$
    **return** $\textsc{Crypt}(P, \text{false})$
```

Figure 13. *Encrypt* Function Definition. Source: [5].

d. Decrypt

The only argument for *Decrypt* is the text to be decrypted. It can only be called in keyed mode. It passes the text to *Crypt* as well as a Boolean 1 to indicate decryption vice decryption. Figure 14 gives the formal definition for *Decrypt*.

```
**Interface:** $P \leftarrow \textsc{Decrypt}(C)$, with $\textsc{mode} = \text{keyed}$
    **return** $\textsc{Crypt}(C, \text{true})$
```

Figure 14. *Decrypt* Function Definition. Source: [5].

e. Squeeze

The only argument in *Squeeze* is the desired hash length in bytes. It passes the hash length and the round constant 0x40 to *SqueezeAny*. Figure 15 gives the formal definition for *Squeeze*.

```
**Interface:** $Y \leftarrow \textsc{Squeeze}(\ell)$
    **return** $\textsc{SqueezeAny}(\ell, \text{‘40‘ (squeeze)})$
```

Figure 15. *Squeeze* Function Definition. Source: [5].

f. SqueezeKey

The only argument in *SqueezeKey* is the desired hash length in bytes. It can only be called in keyed mode. It passes the hash length and the round constant 0x20 to *SqueezeAny* and exists to generate a key. Figure 16 gives the formal definition for *SqueezeKey*.

```
Interface: $Y \leftarrow \textsc{SqueezeKey}(\ell)$, with $\textsc{mode} = \text{keyed}$
  return $\textsc{SqueezeAny}(\ell, \text{‘20‘ (key)})$
```

Figure 16. *SqueezeKey* Function Definition. Source: [5].

g. *Ratchet*

Ratchet accepts no input. It can only be called in keyed mode. The first argument supplied to *AbsorbAny* is the first $l_{ratchet}$ bytes of the permuted *State*. Behaviorally, the series of function calls permutes the state with *f*, then overwrites the $l_{ratchet}$ most significant bytes in the state with zero. Figure 17 gives the formal definition for *Ratchet*.

```
Interface: $\textsc{Ratchet}()$, with $\textsc{mode} = \text{keyed}$
  $\textsc{AbsorbAny}(\textsc{SqueezeAny}(\ell_{\text{ratchet}}, \text{‘10‘ (ratchet)}), R_{\text{absorb}}, \text{‘00‘})$
```

Figure 17. *Ratchet* Function Definition. Source: [5].

4. Top Level Instance

The instance of Xoodyak is determined by a call to the *Cyclist* function. Initially, state is instanced to be all zeros, and the phase is instanced as up. The mode is set to hash, and R_{hash} is assigned to be the R_{absorb} and $R_{squeeze}$ parameter. In keyed mode, these values are overwritten in the call to *AbsorbKey*, but if there is no supplied key then the state remains in hash mode. Figure18 gives the *Cyclist* formal definition.

```
Algorithm 2 Definition of $\textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}]$
Instantiation: cyclist $\leftarrow \textsc{Cyclist}[f, R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}, \ell_{\text{ratchet}}](K, \text{id}, \text{counter})$
  Phase and state: $(\textsc{phase}, s) \leftarrow (\text{up}, \text{‘00‘}^{b'})$
  Mode and absorb rate: $(\textsc{mode}, R_{\text{absorb}}, R_{\text{squeeze}}) \leftarrow (\text{hash}, R_{\text{hash}}, R_{\text{hash}})$
  if $K$ not empty then $\textsc{AbsorbKey}(K, \text{id}, \text{counter})$
```

Figure 18. *Cyclist* Function Instantiation. Source: [5].

E. IMPLEMENTATION

This research implemented the ability to invoke every function call with the opcodes described in the level 1 functions. Function calls are made by supplying an operational code (opcode) to the input ports with the accompanying data. All opcodes are four bits long and expressed in hexadecimal. For example, the hardware will enter the Cyclist function in keyed mode by ingesting opcode 0x01, and *Cyclist* in hash mode by ingesting opcode 0x09.

Broadly speaking, our hardware structure is built around the level 2 functions, the inputs for which are modified depending on the exact function called. Every function except for *Split* is allocated hardware, which is administered by the user. The hardware structure is not able to recognize where one string of information begins or ends, and it is only capable of processing a finite amount of data on every function call. Since this split in data must occur regardless of how long or short a string is, the user is responsible for keeping track of when strings begin or end.[8]

1. Instance Parameters

The Xoodyak Implementation operates as a series of function calls. The Xoodyak algorithm allows for some variable length fields, such as the message text. This implementation used fixed length vector lengths based on parameters described in [5] and summarized in Table 6.

[8] However, no organization is required on the part of the user apart from properly aligning the data input vector and opcodes. Subsequent calls for the same function {Absorb(X), Absorb(Y)} can be discriminated by inserting a single idle opcode for one clock (on a clock that the hardware is accepting inputs). Without this step, the hardware will process the data as {Absorb(X || Y)}, which is not the same operation because of color bytes and ensuing avalanche effect in the *Permute* hardware.

Table 6. Xoodyak Parameter Length Summary. Adapted from [5]

Object	Length (Bytes)	Length (bits)
R_{absorb} (hash)	16	128
R_{absorb} (AEAD)	44	352
M_{length}	24	192
Key Size	16	128
Nonce	16	128
R_{sqz}	16	128

2. ***State*** **Machines**

Operations are governed through three distinct finite state machines. The first state machine determines whether the state is in hash or keyed mode. The second machine handles function calls. The final state machine handles the "*Shadow State*" which stores the previous function call. The *Shadow State* is important for administering color bytes. The state machine and "shadow state machine" can proceed directly from one function call to another. Returning to the idle state will wipe the shadow state, allowing for an identical function with a broken string. Table 7 summarizes valid opcode combinations.

Table 7. Xoodyak Opcode Decode and Mode Restriction.

Function Call	Opcode	Keyed Mode Only
Cyclist (keyed)	1	-
Cyclist (hash)	9	-
Nonce	2	YES
Absorb	3	-
Encrypt	4	YES
Decrypt	5	YES
Squeeze	6	-
Ratchet	7	YES
SqueezeKey	8	YES
Idle	All Others	-

The "Nonce" function performs an *Absorb* with a different parameter length, per Table 6.

3. User Operation of Xoodyak

Xoodyak completes functions in either one clock, four clocks, or 12 clocks depending on the hardware build and the function call. These distinctions are discussed in the next section. These counts do not include startup queuing delays.[9]

a. Startup Delays and Timing

There is an additional two-clock startup delay to sample and alter the state machine. Therefore, a function call completes three, six, or 14 clocks after a function call is queued at the front of the module.[10] This means that in all cases the next function will be sampled while the previous function is still running, unless the state machine is in the idle state.

Any function that uses the *Permute* module completes in four or 12 clocks. The four or 12 clocks required for operation depends on the build used.[11] Both approaches fully satisfy the round requirement levied by the Xoodyak algorithm.

Not every function call requires *Permute,* which triggers a one clock operation. Functions that end with an "UP" state as defined by [5] cause the next function called to not use *Permute*. Multiple calls to the same function do not trigger this case. For example, generating a 256-bit digest requires two sequential calls to *Squeeze*. Each call requires *Permute*, and two calls are required, for a total of ten clocks to generate the entire string.[12] A following function such as *Absorb* (or other) will not require *Permute* as such the next input text and opcode will be sampled on the very next clock. Tables 8, 9, and 10 summarize stream clock delays based on function calls.

[9] All modules support streaming operations. Accordingly, startup delays are not included in any clock related description outside Ch III.E.3.a, Startup Delays, unless explicitly stated otherwise. CH IV.C.3 refers.

[10] Alternatively, two, five, or 13 clocks after the sampling register. That is, if the sampling clock is not counted in the total. If applicable, output texts are provided synchronously on this clock, which is two, five, or 13 clocks after the sampling.

[11] Ch III.E.4.b describes how these build differences impact the required clocks. Appendix E contains the full list of builds. Tables 13–16 contain selected builds with both build versions.

[12] For the four-clock build: a two-clock startup delay, followed by four clocks of operation for the first call. The subsequent call to Squeeze only requires four clocks through streaming operations, having been sampled on clock 5 while the first Squeeze was operating. Clock 10 generates the last set of output text.

Table 8. Functions with 1-Clock Stream Delays.

Function Call	Opcode
Cyclist (keyed)	1
Cyclist (hash)	9

Table 9. Functions That Make the Next Non-Identical Function Have a 1-Clock Stream Delay.

Function Call	Opcode
Cyclist (hash)	9
Squeeze	6
SqueezeKey	8

Table 10. Functions with a 4-Clock or 12-clock Stream Delay by Default

Function Call	Opcode
Nonce	2
Absorb	3
Encrypt	4
Decrypt	5
Squeeze	6
Ratchet	7
SqueezeKey	8

4. Structural Datapath

There are three parts to the datapath, Pre-*Permute*, *Permute*, and Post-*Permute*. Each section loosely relates to different stages in the level 2 functions. Hardware in the Pre-*Permute* module performs exception handling. Pre-*Permute, Permute,* and Post-*Permute* collectively perform all functions. The user performs *Split* on the data, but the hardware will track whether a function call is continuous. The user can indicate a string for a function by continuously asserting the same opcode when the hardware completes function calls. This allows the user to handle arbitrary length function calls, provided those lengths are multiples of the parameters specified. The user can provide padding to meet those multiples.

a. Pre-Permute

The purpose of the Pre-*Permute* stage is to generate the 384-bit vector that enters the *Permute* module. Figure 19 provides a high-level process overview of the hardware structure that handles the per *Permute* stage. By default, the *State* is the contents of a 384-bit vector which is the output of the previous function call. There are several exceptions, and the Pre-*Permute* stage handles all of them in a multiplexer. The exceptions are summarized in the next section. Multiple exceptions cannot occur at the same time, so dedicated exception handling hardware is reused in several cases.

A round constant is then applied to the output, depending on the function call, as described in Algorithm 2 [5]. The round constant is only added for the first permutation of a string.

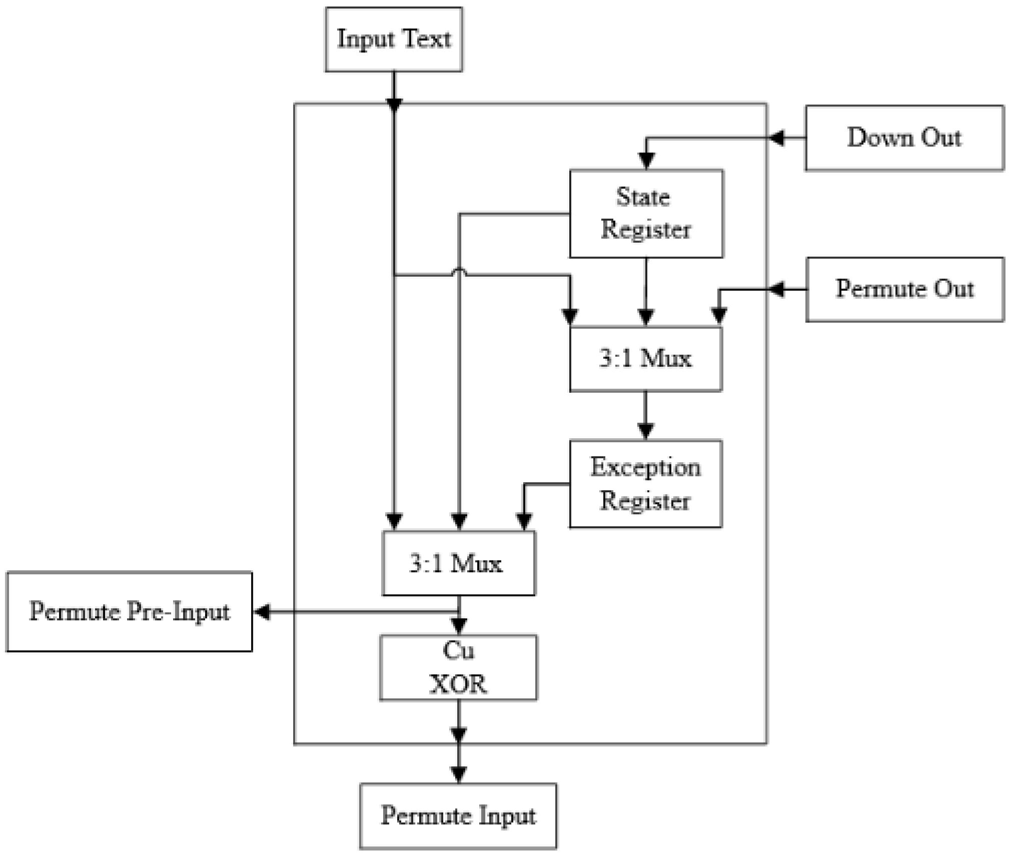


Figure 19. Pre-*Permute* Process Diagram

b. Exception handling

There are two exceptions to handle edge cases. The cause of the edge cases is that the hardware does not track the algorithmic phase - “UP” or “DOWN.” Rather, each pass through the hardware accomplishes both the *Up* and *Down* functions, which creates issues when one of those functions is not used. There are two ways this can occur. The first way is after a *Squeeze* function. Recalling Figure 16, see that regardless of the string length *l*, the phase ends in the “UP” state at the completion of the function call. Since the user is performing the *Split* function, the user delimits the end of the *Squeeze* function by simply providing the next function and associated data. Inside the hardware, the *Shadow State* is *shadow_squeeze*, and the actual state machine is not in the *Squeeze* state, which triggers the exception. The impact is that the next function call, delimited by the next ingested opcode other than 4’h6 (*Squeeze*) or 4’h8 (*SqueezeKey*), will not perform the *Up* function and associated *Permute.* Additionally, the value held in the *State* register will be invalid, and the value in the exception register will be used instead. We speculate this exception will not be especially relevant, since performing a *Squeeze* or *SqueezeKey* is a hash of the existing *State*, which implies the operation is complete. However, it is included for completeness of operation.

The second exception is in the first *Absorb* function call after *State* instantiation with opcode 4’h9, *Cyclist* (hash). Unlike the hardware keyed mode *Cyclist* function, which also performs the algorithmic *AbsorbKey* function, the hardware hash *Cyclist* function merely only creates a 384-bit vector of all zeros. Accordingly, the first *Absorb* after this only requires an algorithmic *Down* function and is completed in a single clock.[13]

[13] We considered including the first hash-based *Absorb* in the *Cyclist* instantiation, requiring the user to supply the first round of *Absorb* data at instantiation. This would theoretically cause a one clock advancement of any function sequence, but we decided against it because this would be combining two different top-level functions at once, which is not consistent with the proposed standard [5]. This is directly in opposition to the associated key absorption innate in the keyed *Cyclist* function, which directly calls the first *Absorb* without further top level involvement and therefore does not trigger an exception under our rules.

c. *Permute*

The *Permute* module is made up of three parts, a round, round constant vector, and a register. The *Permute* inputs are fed to the *Permute* module and registered every clock. Figure 20 provides a high-level view of this process. Multiple *Permute* rounds can be instanced in sequence. This research specifically built one-*Permute* instances and three-*Permute* instances.[14] 12 rounds must be accomplished in total, and the addition of multiple *Permute* instances reduces the number of clocks required to complete a permutation.

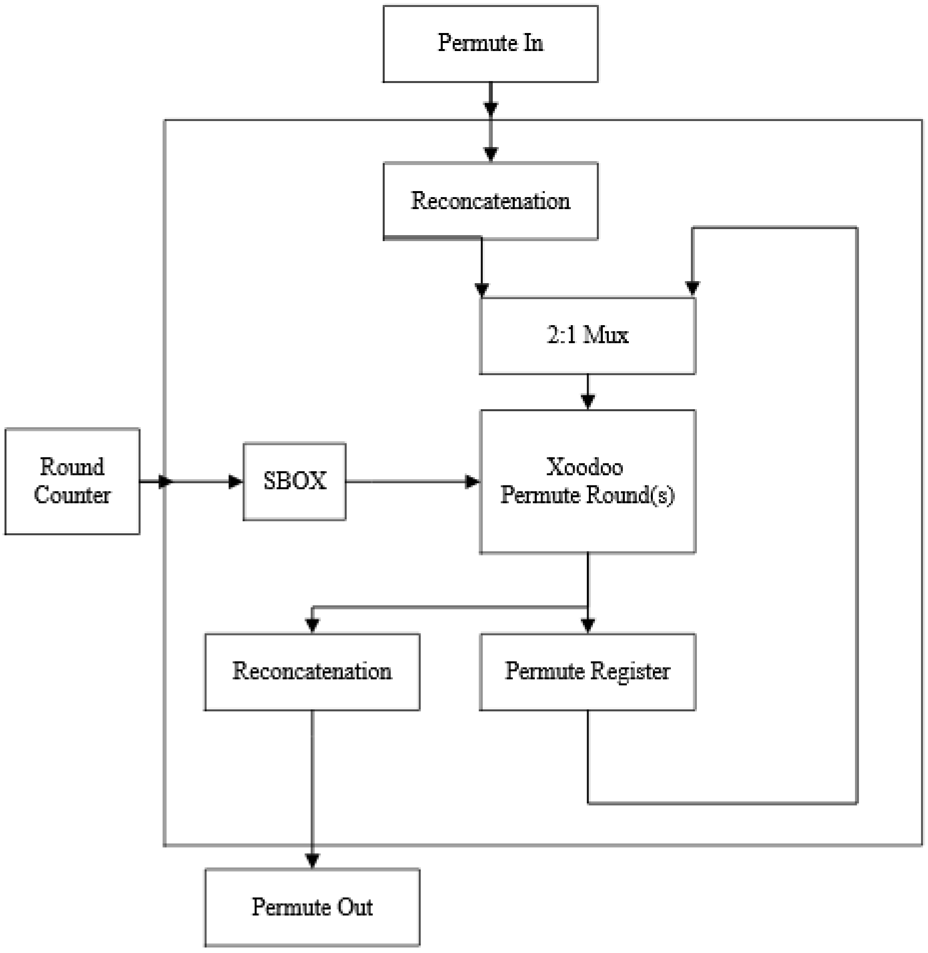
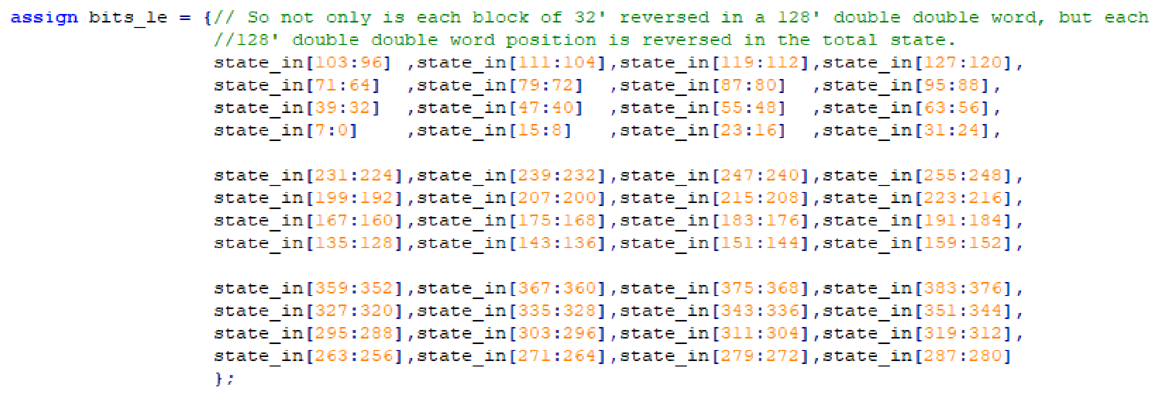


Figure 20. *Permute* Process Diagram

[14] See CH IV for build analysis. See Appendix E for full list of builds.

Execution of a *Permute* round is exactly as described in [5], with a noted exception in the way certain indices are referenced. The test vector and software indexing are reversed compared to SystemVerilog conventions. This drastically changes *Permute* results. Figure 21 is a direct excerpt from the RTL code that shows this reversal.



```
assign bits_le = {// So not only is each block of 32' reversed in a 128' double double word, but each
                  //128' double double word position is reversed in the total state.
                  state_in[103:96] ,state_in[111:104],state_in[119:112],state_in[127:120],
                  state_in[71:64]  ,state_in[79:72]  ,state_in[87:80]  ,state_in[95:88],
                  state_in[39:32]  ,state_in[47:40]  ,state_in[55:48]  ,state_in[63:56],
                  state_in[7:0]    ,state_in[15:8]   ,state_in[23:16]  ,state_in[31:24],

                  state_in[231:224],state_in[239:232],state_in[247:240],state_in[255:248],
                  state_in[199:192],state_in[207:200],state_in[215:208],state_in[223:216],
                  state_in[167:160],state_in[175:168],state_in[183:176],state_in[191:184],
                  state_in[135:128],state_in[143:136],state_in[151:144],state_in[159:152],

                  state_in[359:352],state_in[367:360],state_in[375:368],state_in[383:376],
                  state_in[327:320],state_in[335:328],state_in[343:336],state_in[351:344],
                  state_in[295:288],state_in[303:296],state_in[311:304],state_in[319:312],
                  state_in[263:256],state_in[271:264],state_in[279:272],state_in[287:280]
                  };
```

Figure 21. Reconcatenation Logic

d. Post Permute.

The output of the *Permute* module is unregistered; the register is the beginning of the intermediate rounds rather than the end. The vector subject to post-permute operations depends on the function presented. Not every function requires a permute. Functions that require *Permute* use the outputs of *Permute*. Functions that do not require *Permute* use the *Permute* input vector before the round constant is added.

The *Down* function is computed for every type of function in parallel. The final output is selected via a multiplexer controlled by the *State* Machine. Since several functions are similar in form, each *Down* operator can accomplish multiple function variants. Figure 22 provides a high level view of this process, from input to output data.

Output text is only valid for certain function calls. For these functions, a multiplexer performs a modification of the *Down* output to generate the output text.

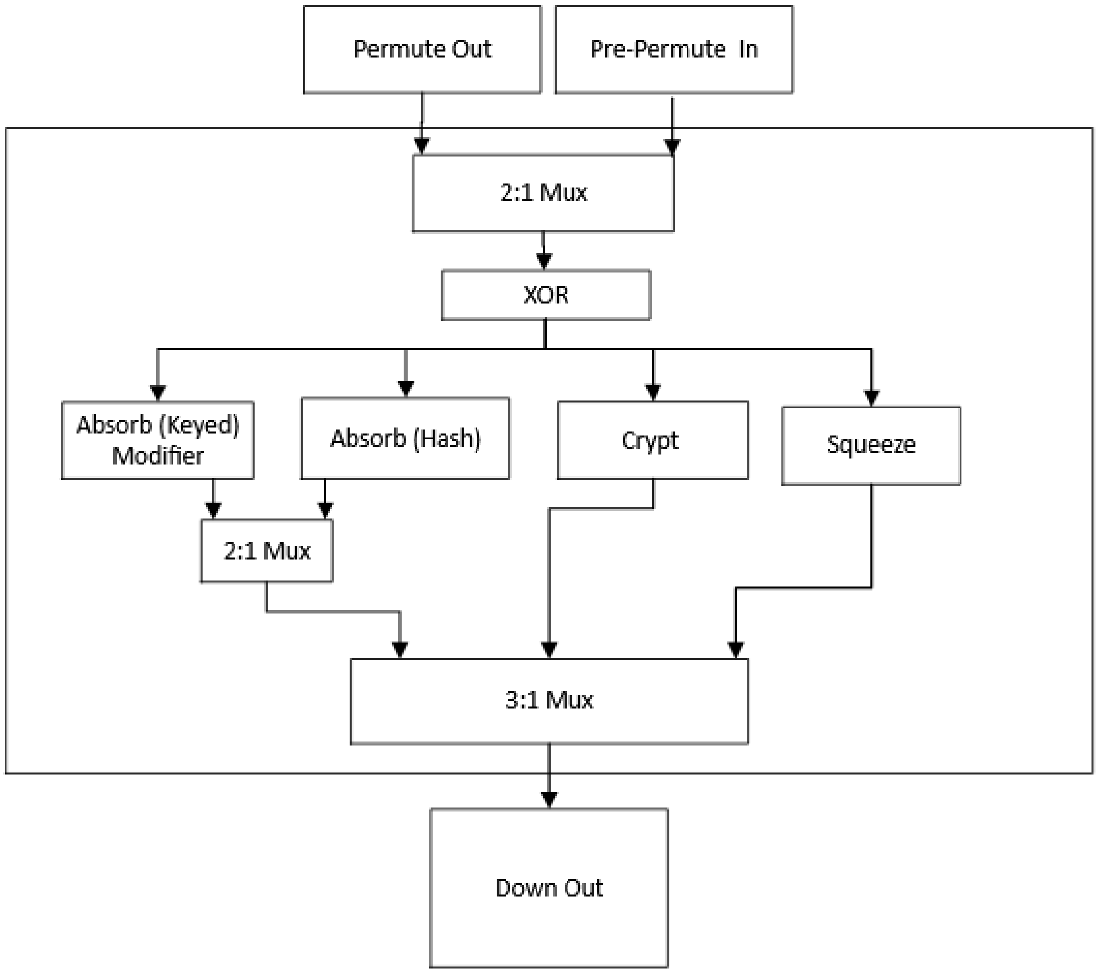


Figure 22. Post-*Permute* Process Diagram

5. Verification

The Xoodyak implementation supports nine function calls and idle. Verification was performed by hand for timing, function returns, and data manipulation. Further verification was performed via Sean B. Palmer's software implementation "Xoocycle," published as part of the NIST submission package [8]. Vectors were hand-examined via Xoocycle and Vivado waveform simulation. The waveform window was examined for expected output text, output decode, and timing across a wide variety of function calls. In total, each callable function was performed and checked for accuracy, including using strings that are too long to process in a single operation. Figure 23 shows a subjective view of the process. Appendix C shows an example of a sequence of function calls, complete with input data and *State* values between each function.

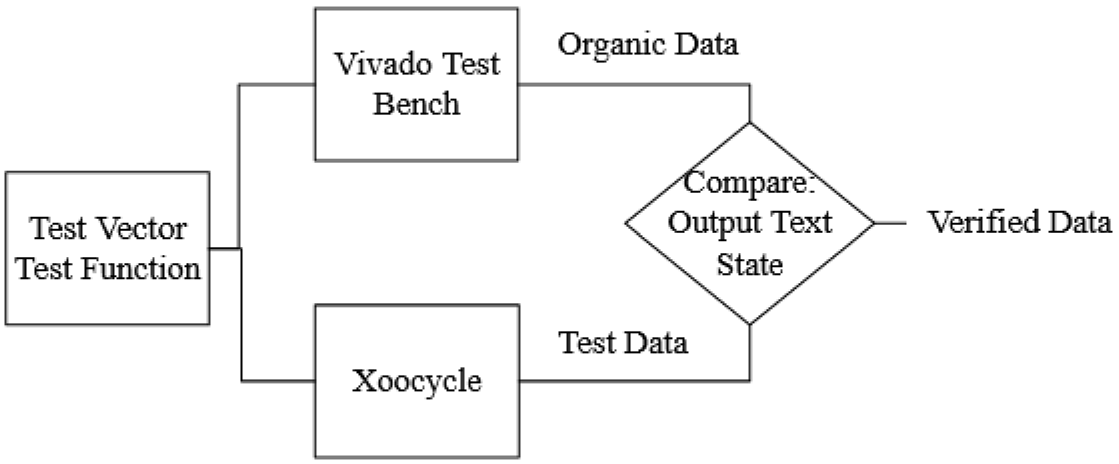


Figure 23. Xoodyak Data Verification Process

F. SUMMARY

This chapter examined the algorithmic definition, hardware structure, and verification process for Xoodyak. The RTL code which implements Xoodyak is an operational module capable of arbitrary length and number of function calls, subject to certain parameters. The Xoodyak module has also survived non-exhaustive verification by hand. The instance as implemented forms a thorough basis for detailed performance and cell area comparisons with our AES-128 module. These detailed comparisons are the topic of the next chapter.

IV. RESULTS

Our comparison between AES-128 and Xoodyak has three main dimensions: functionality, throughput, and size. The build data does not include dynamic power, which is important for some applications. However, leakage power is included in the build data.

Direct comparisons between our AES-128 and Xoodyak requires certain rules. These include baseline assumptions on what qualifies a build for a detailed comparison, or a subjective assessment on which builds are more competitive than others. It also includes basic benchmarking rules, such as the decision to use stream encryption as the primary throughput benchmark, and the use of cell area vice die area as the primary size metric.

Functionality is assessed on support for cryptographic capabilities which include authentication, encryption, hashing, and associated data support. Throughput is assessed in bits processed per second. Size is assessed in terms of cell area. Lastly, there is a brief discussion on alternative comparisons between the algorithms which were not implemented.

A. COMPARATIVE FUNCTIONALITY ANALYSIS

Our AES build can support full AEAD and hashing through the NIST recommendation [3]. However, AES as described in the AES Standard is only capable of block encryption and decryption [1]. The Xoodyak suite includes full AEAD and hashing support by default. All Xoodyak builds in this research implement the full Xoodyak suite, as summarized in Table 11.

Table 11. Function Comparison - Xoodyak and AES-128

Functions	
Xoodyak	AES
Encrypt	KeyExpansion
Decrypt	Encrypt
Absorb	Decrypt
Squeeze	No equiv.
SqueezeKey	No equiv.
Ratchet	No equiv.

The Xoodyak suite, when combined in the manner specified by Daemen et al., fully implements the NIST requirement for AEAD and hashing [5].[15] Function calls can take place in any arbitrary sequence as defined by the user.[16] The actual capability delivered by Xoodyak is therefore highly flexible and exceeds the minimum NIST AEAD and hash requirements. There is no loss in claimed security with any of these builds. Both AES-128 and the Xoodyak implementation require 128-bit keys and claim 128 bits of security.[17]

B. SUBJECTIVE STRUCTURAL ANALYSIS

AES and Xoodyak have distinct hardware structures. AES is defined by rounds; each round uses a *RoundKey* generated by *KeyExpansion* and is comprised of four well-defined mathematical steps in sequence. The round calculation is repeated 10 times in AES-128. The calculations are applied directly to the plaintext or ciphertext, along with the expanded keys.

The main cryptographic operation in Xoodyak is the *Xoodoo*/*Permute* function. The permutation is applied to the *State*, not the input text. The value of the *State* depends on the process history, including the order of function calls and data absorbed since *State* instantiation. The text is applied via XOR after 12 permute rounds, like an AES

[15] Algorithmic use of Xoodyak in AEAD mode is described in Section 3.2.3 – Authenticated Encryption [5]. Use of hash mode is described in Section 3.1 – Hash mode.

[16] The Xoodyak algorithm and implementation can handle any arbitrary sequence of function calls. However, an arbitrary series of calls is not guaranteed to be a valid use of AEAD/hash or other cryptographic significance.

[17] The NIST competition submission algorithm uses a 128-bit key as a parameter. Instantiating a Xoodyak state requires the use of a 128-bit key in the implementation. A longer key can be appended through the Xoodyak *Absorb* function. The additional security of such an addition is not evaluated.

AddRoundKey.[18] Substantial combinational logic outside permute is required to support function calls. Every function modifies the *State* before permutation, and further modifications are applied after *Permut*e.

There are two implemented structures of Xoodyak builds. The differences in the builds stem from the way they handle the permute function. Xoodyak X-05N and X-07N series builds require 12 clocks to complete. Xoodyak X-06N and X-08N series builds accomplish three permute rounds in one clock and thus require only four clocks to complete.

This improved throughput performance comes at the cost of additional hardware. Jim Donahue at Centaur Technology Inc. built both the AES and Xoodyak variants [9]. Builds were done in both a 16nm technology and a 5nm technology to identify any algorithm related throughput, size, or leakage current differences due to technology.

The build recipes were similar to those used by Centaur for internal designs. These builds are complete for a design decision usage, but do not have the final tapeout adjustments including design rule check fixes, hold time fixes, or pad cells.

C. IMPORTANT CONSIDERATIONS FOR BENCHMARKING

1. Build Qualification for Benchmarking

A total of 26 build variations were generated between AES-128 and Xoodyak. Comparing each build was not feasible. The intent of the various build configurations was to find the fastest realistic frequency for a design and to perform design comparisons at similar clock frequencies. From those build results, comparisons can be drawn for the most competitive builds with a certain baseline. The baseline requirements for detailed comparison were a clock frequency above 2 GHz, and a leakage current lower than 1mW.

[18] The nature of the input text depends on the function called, whether plaintext, associated data, or otherwise.

2. Precision Limitations on Results

While official build results are highly specific, they are not necessarily precise. The build results should be interpreted as working examples rather than sole truths. Multiple builds of the same assumptions and RTL code could have variable performance. Variability in clock speed could be as high as +/- 50MHz. Tabulated results therefore round results at 100MHz. The full results in Appendix E do not round these values.

3. Startup Delays, Streaming Mode, and Performance

Both the AES-128 and Xoodyak implementations require startup delays before function execution. AES-128 requires a one-clock delay. Xoodyak requires a two-clock delay. These delays exist to register input data and opcodes and initialize state machines. Non-streaming operations are required to account for these delays, but streaming operations are not.

All implementations support streaming mode. Startup delays can be eliminated for streaming operations because the modules will accept new input data while a previous operation is ongoing.

For example, the AES-128 *Encrypt* and *Decrypt* modules carry a 11-clock delay between input sample to output text generation. However, the state machine will allow the next text vector to be registered on clock 10. The startup clock cycles are paid in parallel with the previous encrypt cycle. The streaming delay for AES *Encrypt* and *Decrypt* is therefore 10 clocks rather than 11. Streaming startup delays in Xoodyak can be ignored for the same reason.

An arbitrary number of encryptions can be performed with only one two-clock startup delay under this construct. This is insignificant across a large operation. Accordingly, they are neglected in streaming throughput computations.

4. Adjustments for AES *KeyExpansion*

The AES instance includes a *KeyExpansion* module. For ease of implementation, keys are not generated "on the fly." Key generation requires a 11-clock delay from input to output register. Combined with the 10-clock stream delay for *Encrypt* and *Decrypt* modules, this means there is a 21-clock delay from the point a true key is supplied to generate output text using that key. However, when computing the throughput performance for AES the *KeyExpansion* clocks are not included.

D. PARAMETER ANALYSIS

All instances of Xoodyak included the full Xoodyak suite with both AEAD and hash functionality. Tables 12 – 17 provide an overview of the throughput, size, and leakage power results for certain builds. While all tables contain every parameter, builds in Tables 12–15 are selected for throughput analysis and builds in tables 16–17 are selected for cell area analysis. When referenced by build, A-series builds refer to AES-128 and X-series builds refer to Xoodyak. Builds with 5nm technology, vice 16nm technology, can be identified by the "5n" in the build name. Appendix E contains the full raw results, and the following subsections contain the relevant discussion.

Table 12. Streaming Results—5nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-5n03d	2,800	0.83	4.0	10	128	51
Xoodyak	X-5n06a	2,200	0.86	2.9	4	192	140
Xoodyak	X-5n05g	2,100	0.77	5.1	12	192	82

Table 13. Streaming Results—16nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-03c	7,400	0.81	2.4	10	128	31
Xoodyak	X-08a	6,300	2.50	2.2	4	192	106
Xoodyak	X-07d	4,200	0.78	3.4	12	192	54

Table 14. Non-Streaming Results—5nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-5n03d	2,800	0.83	4.0	11	128	47
Xoodyak	X-5n06a	2,200	0.86	2.9	6	192	93
Xoodyak	X-5n05g	2,100	0.77	5.1	14	192	70

Table 15. Cell Area Results—16nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-03c	7,400	0.81	2.4	11	128	28
Xoodyak	X-07b	3,300	0.23	2.4	14	192	33
Xoodyak	X-08a	6,300	2.50	2.2	6	192	70
Xoodyak	X-07d	4,200	0.78	3.4	14	192	47

Table 16. Cell Area Results - 5nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-5n03d	2,800	0.83	4.0	10	128	51
AES-128	A-5n03f	2,200	0.20	2.1	10	128	27
Xoodyak	X-5n05c	1,200	0.25	3.4	12	192	54
Xoodyak	X-5n05e	1,600	0.44	5.0	12	192	80
Xoodyak	X-5n05g	2,100	0.77	5.1	12	192	82
Xoodyak	X-5n06a	2,200	0.86	2.9	4	192	140
Xoodyak	X-5n06b	1,900	0.64	2.6	4	192	125

Table 17. Cell Area Results - 16nm Technology. Adapted from [9]

Algorithm	Build #	Cell Area	Leakage	Clock		Width	Thruput
-	-	μm^2	mW	GHz.	Clks/Op	Bits/op	Gbits/s
AES-128	A-03c	7,400	0.81	2.4	10	128	31
AES-128	A-03d	7,100	0.46	2.1	10	128	27
AES-128	A-03f	11,300	7.47	3.6	10	128	46
Xoodyak	X-07b	3,300	0.23	2.4	14	192	33
Xoodyak	X-07c	3,500	0.43	2.9	12	192	46
Xoodyak	X-07d	4,200	0.78	3.4	12	192	54

1. Throughput Analysis

X-5n06a has the highest throughput of any build. The high throughput value stems from the three *Permute* rounds instantiated in series. This allows any hardware stream function call, including *Encrypt* and *Decrypt*, to be completed in four clocks.[19] X-6n06a strikes the best balance between clock speed and clock count to achieve the highest possible throughput with a feasible clock rate.

Per Table 14, X-5n06a has a stream throughput factor of 2.7 times A-5n03d, the highest-throughput AES-128 build. This throughput factor includes a 192-bit data width for Xoodyak. Normalizing Xoodyak performance at 128-bits of true data per function cycle requires a downward revision to 1.8.

Performance gains at the 16nm technology level are less decisive. Build X-08a is the highest throughput 16nm Xoodyak build with a throughput value of 106 Gbit/s, but its leakage current value is 2.5mW— far above the soft 1mW specification. This also means that there were no qualified three-*Permute* 16nm builds. Hence, we move to the one-*Permute* 16nm instances for comparisons to AES.

The highest throughput 16nm build with less than a 1mW leakage is X-07d. X-07d is a one-*Permute* build with half the throughput of X-08a, its three *Permute* competitor. X-07d does claim a throughput factor of 1.74 over AES build A-03c, but this includes a wider Xoodyak data width. Normalizing to a 128-bit data width for Xoodyak reduces that factor to 1.16.

2. Cell Area Analysis

The various AES-128 and Xoodyak instances were built using the ASIC flow build process described in Appendix D. Die plots are shown in Appendix F. While every build has a die size, the isolated builds are not reflective of the die size in a real chip. For example, all inputs and outputs are registered in every research build. This may not be necessary if

[19] All X-series (Xoodyak) builds with a 4 or 6 in the "Clks/Op" column has three permute rounds instantiated in series. X-series builds that require 12 or 14 clocks have one permute round only. Broadly speaking, builds with three permute rounds were intended to maximize throughput and builds with one permute round were intended to minimize cell area.

these modules were built within a larger chip because the input vector may be able to make timing without an input register. The surrounding circuitry context is required to determine this possibility.

Also, a real chip may have more accessible layers which reduces the required die area footprint. “Cell area” is the sum of all the area of all combinational logic and registers. Cell area is therefore unaffected by this circuit context. Accordingly, “cell area” as opposed to “die area” was the evaluated logic size metric.

X-5n05e and X-07b contain the lowest cell area for their respective technologies, with cell areas of 1,600μm^2 and 3,300μm^2 respectively. Compared with the most cell-area-efficient AES-128 builds, Xoodyak provides a 28% cell area reduction with 5nm technology and a 53% reduction with 16nm technology. These reductions were achieved by the instance of a single *Permute* round, which necessitated a 12-clock function call to achieve the necessary 12 rounds. The green and white cells in Figures 25 and 26, which represent *Permute* hardware, illustrate the impact of having a single *Permute* instance versus three in series.

Tables 17 and 18 also empirically demonstrate the positive relationship between operational frequency, cell area, and leakage current. This difficult balance is best shown in the differences between builds A-03c and A-07f, two AES-128 builds where A-07f generated the highest throughput of any AES-128 build. Unfortunately, the circuitry required to make this timing meant that the cell area requirements was 11,300μm^2, and the 7.47mW leakage current value meant that this build was unqualified for a throughput benchmark under our rules.

E. ALTERNATIVE COMPARISONS

1. Alternative: Xoodyak and AES-Galois Counter Mode (AES-GCM)

The full Xoodyak suite has full AEAD functionality. The AES core does not. Ref [3] recommends AES-GCM for AEAD and is in common use but is not a formal NIST standard. AES-GCM is a more functionally direct comparison to Xoodyak. Implementing AES-GCM was beyond the scope of this research, but AES-GCM would have required additional cell area, and possibly lower throughput, to provide total equivalent functionality to Xoodyak.

2. Alternative: Enc/Dec Xoodyak only and AES Core

A Xoodyak instance with only the *AbsorbKey*, *Encrypt* and *Decrypt* functions defined may be a more functionally-direct comparison to the AES core. Limited functionality might reduce the cell area required and increase clock frequency, improving the comparisons given in Tables 13–18. However, the full permute instance is required regardless of functionality because every function uses the same permute function. Large performance gains, and size and power reductions associated with these functionality reductions are unlikely.

F. SUMMARY

This chapter discussed the performance comparisons of our AES-128 and Xoodyak modules, as built by Jim Donahue of Centaur Technologies. The results show that Xoodyak builds optimized for throughput can achieve approximately 3:1 throughput gains over AES-128 for both 5nm and 16nm ASIC technology. Results also show that Xoodyak builds optimized for cell area occupy approximately half of the cell area required by AES-128 for 16nm technology, and are 28% smaller for 5nm technology. Appendix E contains a full readout of all 26 discrete builds.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

The NIST Lightweight Cryptography Competition created interest in lightweight cryptographic algorithms. Examination of proposed standards involved a comparison to AES, the established standard. Our research benchmarked AES-128, as a standard, against Xoodyak, the proposed lightweight cryptography standard. RTL code was written to support these implementations, which was then built by Centaur Technologies.

A. ASSESSMENT OF GOALS

Our results showed that selected Xoodyak modules outperformed AES-128 in direct comparisons, while also demonstrating AEAD and hashing support. For Xoodyak builds optimized for performance, results showed in a throughput factor of 2.7 for 5nm technology and 1.7 for 16nm technology. Likewise, Xoodyak builds optimized for cell area carry a 28% reduction in 5nm technology, and a 53% reduction in 16nm technology. These results do not guarantee identical gains when integrated, but they provide a valuable benchmark of relative strength when written by the same author and built by the same tools at Centaur. Our work was successful in quantifying these advantages and provides additional information to inform the NIST competition.

B. FUTURE WORK

1. Remove Fixed Length String Requirement

One of the assumptions made in the hardware is that every input vector is presumed to be the same length – specifically, the maximum length given by Xoodyak parameters. This simplified debugging and may have reduced the hardware requirement. The algorithm does not require input strings to be even multiples of the parameter lengths. An additional input vector specifying the length of the input string may be enough to fully capture this fidelity. Since the largest *Split* parameter in Xoodyak is in *Absorb* with a value of 352 bits, an additional 6-bit vector is large enough to specify any parameter length in Xoodyak.

2. Perform a More Extensive Verification

Xoodyak is a new algorithm and test vectors are difficult to come by. The verification methods used in this work were therefore quite rudimentary and primarily involved pen and paper verification and waveform inspection on a bit-by-bit and clock-by-clock basis. While a non-trivial number of vectors and function calls were tested, we were limited to discretely checking only a very small number of functions, such as those specifically listed in Figures 4 and 5. More robust methods would be required if these modules were to enter production.

3. Reduce the Xoodyak Startup Clocks from Two to One

Xoodyak requires two startup clocks compared to one for AES as discussed in Chapters III and IV. We believe this can be reduced to one through a modification of the state machine and associated registers. This reduction would not improve streaming throughput, but it would improve "one off" throughput and further optimize the code.

4. Perform an ASIC Benchmarking of AES with Galois Counter Mode

We performed standard to proposed standard comparison between AES and Xoodyak. This may have been unfair to Xoodyak, as a functionally equivalent comparison with AES should include AES with the NIST recommendation for Galois Counter Mode. It is certain that AES-GCM will occupy more cell area than the AES core as implemented, and it is possible that the throughput will be negatively affected as well. If this is the case, then the throughput and size factors computed in this work are in fact lower bounds.

APPENDIX A. CODE REPOSITORY

Open-source AES code repository.
GitHub: https://github.com/MikeWakeland/AES-SV
POCs: Michael.C.Wakeland@gmail.com, ccw@nps.edu

Open-source Xoodyak code repository.
GitHub: https://github.com/MikeWakeland/Xoodyak-HW/tree/main/Xoodyak_functions
POCs: Michael.C.Wakeland@gmail.com, ccw@nps.edu

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B, DEFINITION OF XOODOO

The Xoodoo definition is an excerpt from “Xoodyak, a lightweight cryptography scheme.” Source: [5].

2.1 The Xoodoo permutation

XOODOO is a family of permutations parameterized by its number of rounds n_r and denoted XOODOO$[n_r]$.

XOODOO has a classical iterated structure: It iteratively applies a round function to a state. The state consists of 3 equally sized horizontal *planes*, each one consisting of 4 parallel 32-bit *lanes*. Similarly, the state can be seen as a set of 128 *columns* of 3 bits, arranged in a 4×32 array. The planes are indexed by y, with plane $y = 0$ at the bottom and plane $y = 2$ at the top. Within a lane, we index bits with z. The lanes within a plane are indexed by x, so the position of a lane in the state is determined by the two coordinates (x, y). The bits of the state are indexed by (x, y, z) and the columns by (x, z). *Sheets* are the arrays of three lanes on top of each other and they are indexed by x. The XOODOO state is illustrated in Figure 1.

The permutation consists of the iteration of a round function $\mathrm{R_i}$ that has 5 steps: a mixing layer θ, a plane shifting ρ_{west}, the addition of round constants ι, a non-linear layer χ and another plane shifting ρ_{east}.

We specify XOODOO in Algorithm 1, completely in terms of operations on planes and use thereby the notational conventions we specify in Table 1. We illustrate the step mappings in a series of figures: the χ operation in Figure 2, the θ operation in Figure 3, the ρ_{east} and ρ_{west} operations in Figure 4.

The round constants C_i are planes with a single non-zero lane at $x = 0$, denoted as c_i. We specify the value of this lane for indices -11 to 0 in Table 2 and refer to Appendix A for the specification of the round constants for any index.

Finally, in many applications the state must be specified as a 384-bit string s with the bits indexed by i. The mapping from the three-dimensional indexing (x, y, z) and i is given by $i = z + 32(x + 4y)$.

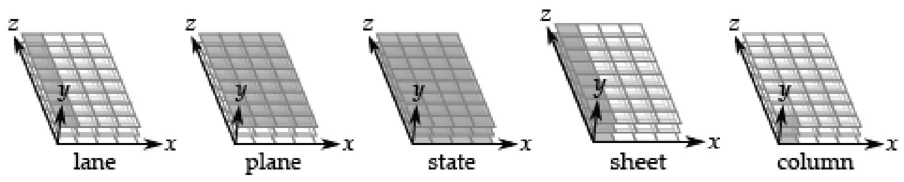


Figure 1: Toy version of the Xoodoo state, with lanes reduced to 8 bits, and different parts of the state highlighted.

Table 1: Notational conventions

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Algorithm 1 Definition of Xoodoo$[n_r]$ with n_r the number of rounds

```
**Parameters:** Number of rounds $n_r$
**for** Round index $i$ from $1 - n_r$ to 0 **do**
  $A = \mathrm{R}_i(A)$

Here $\mathrm{R}_i$ is specified by the following sequence of steps:
    $\theta$ :
        $P \leftarrow A_0 + A_1 + A_2$
        $E \leftarrow P \lll (1, 5) + P \lll (1, 14)$
        $A_y \leftarrow A_y + E$ for $y \in \{0, 1, 2\}$
    $\rho_{\text{west}}$ :
        $A_1 \leftarrow A_1 \lll (1, 0)$
        $A_2 \leftarrow A_2 \lll (0, 11)$
    $\iota$ :
        $A_0 \leftarrow A_0 + C_i$
    $\chi$ :
        $B_0 \leftarrow \overline{A_1} \cdot A_2$
        $B_1 \leftarrow \overline{A_2} \cdot A_0$
        $B_2 \leftarrow \overline{A_0} \cdot A_1$
        $A_y \leftarrow A_y + B_y$ for $y \in \{0, 1, 2\}$
    $\rho_{\text{east}}$ :
        $A_1 \leftarrow A_1 \lll (0, 1)$
        $A_2 \leftarrow A_2 \lll (2, 8)$
```

Table 2: The round constants c_i with $-11 \leq i \leq 0$, in hexadecimal notation (the least significant bit is at $z = 0$).

i	c_i	i	c_i	i	c_i	i	c_i
−11	0x00000058	−8	0x000000D0	−5	0x00000060	−2	0x000000F0
−10	0x00000038	−7	0x00000120	−4	0x0000002C	−1	0x000001A0
−9	0x000003C0	−6	0x00000014	−3	0x00000380	0	0x00000012

APPENDIX C VECTOR EXAMPLE

The Xoodyak implementation correctly performs an arbitrary series of function calls, each of arbitrary length.[20] Long strings must be ingested in a piecemeal fashion as described in the algorithm. This is an example of how:

$$CYCLIST\ (K, id, counter)$$

$$ABSORB(X)$$

$$ENCRYPT(P)$$

$$SQUEEZE(L)$$

K is the 128 bit string:

38393a3b3c3d3e3f3031323334353637

X is the 704 bit string:

6162636465666768696a6b6c6d6e6f706162636465666768696a6b6c6d6e6f706162636465666768696a6b6c

6162636465666768696a6b6c6d6e6f706162636465666768696a6b6c6d6e6f706162636465666768696a6b6c

P is the 352 bit string:

4d4e4f5051525354555657584142434445464748494a4b4c4d4e4f5051525354555657584142434445464748494a4b4c

[20] Subject to individual parameter length requirements - user provides padding.

Function
Cyclist Instance (Keyed Mode) - opcode 1
Registered Input Text [351:0]
38393a3b3c3d3e3f303132333435363700000000000000000000000000000000000000000000000000000000
Input State [383:0]
Undefined
Output State [383:0]
38393a3b3c3d3e3f303132333435363700010000000000000000000000000000000000000000000000000000000000002

Function
Absorb - opcode 3
Registered Input Text [351:0]
6162636465666768696a6b6c6d6e6f706162636465666768696a6b6c6d6e6f706162636465666768696a6b6c
Input State [383:0]
38393a3b3c3d3e3f303132333435363700010000000000000000000000000000000000000000000000000000000000002
Output State [383:0]
0ccb63f23dc3114bcb8c36b63cef99564339ffadac0fbb2f741fe9a5d9b9de250690b87018c5b3b541c0996293e436a8

Function
Absorb - opcode 3
Registered Input Text [351:0]
6162636465666768696a6b6c6d6e6f706162636465666768696a6b6c6d6e6f706162636465666768696a6b6c
Input State [383:0]
0ccb63f23dc3114bcb8c36b63cef99564339ffadac0fbb2f741fe9a5d9b9de250690b87018c5b3b541c0996293e436a8
Output State [383:0]
1b43a7cd6fa2e21dac4688e1d8a7206c023ac58e5af69245303387eefad068bc9f90ee310414057f2d5a18c68a307c4c

Function
Encrypt - opcode 4
Registered Input Text [351:0]
4d4e4f5051525354555657584142434445464748494a4b4c0000000000000000000000000000000000000000
Input State [383:0]
1b43a7cd6fa2e21dac4688e1d8a7206c023ac58e5af69245303387eefad068bc9f90ee310414057f2d5a18c68a307c4c
Output State [383:0]
5a7623842abdf7a08558bc7ad19c93913cb711728192562aed97c6bf7bd9e1547f48ad7dbafef8bf2cc790d1e3b3f9f0

Function
Encrypt - opcode 4
Registered Input Text [351:0]
4d4e4f505152535455565758414243444546474849 4a4b4c0000000000000000000000000000000000000000
Input State [383:0]
5a7623842abdf7a08558bc7ad19c93913cb711728192562aed97c6bf7bd9e1547f48ad7dbafef8bf2cc790d1e3b3f9f0
Output State [383:0]
5bdef9a0659765e8241872a1e59ff766c4fca03d31ace48e1c9a300fc8b57420d4435bbfa2efdb0a01afb869ec8cfa54

Function
Squeeze Key - opcode 8
Registered Input Text [351:0]
Don't Care
Input State [383:0]
5bdef9a0659765e8241872a1e59ff766c4fca03d31ace48e1c9a300fc8b57420d4435bbfa2efdb0a01afb869ec8cfa54
Output State [383:0]
9bd1c096d76a4f742a7f1037f7a697de6ad3187b8ac831a7fceedddb339d5e7a6a4a2f3ce365c931bbd154dde2914c4b

APPENDIX D. AES AND XOODYAK BUILD METHODOLOGY

Adapted from Jim Donahue, Centaur Technologies [9].

```
------------------------------------------------------------------------
BUILD METHODOLOGY                  JimD 2021-10-06
------------------------------------------------------------------------

Technology & Tools
------------------------------------------------------------------------
Tools (all June 2021 release S-2021.06) used:
Synopsys Design Compiler NXT in topographic mode
 - mapping RTL to technology target library of stdcells
Synopsys ICC2
 - import gates from DC-NXT
 - create floorplan with user constraints
 - place gates
 - build clock tree then route & optimize it
 - initial route
 - route optimization
Synopsys StarRC
 - parasitic (RC) extraction
Synopsys Primetime
 - signoff-quality timing tool

Technology used:
TSMC 16nm with 13 layers of metal plus AP top-level
Centaur-designed 12-track stdcell library
AES builds limited to M2-M7 for routing, to keep
 upper layers of metal free for global routes

The flow, meaning the sequence of operations and tcl-scripts which
control it, is a shared Centaur-specific flow that is based on
Synopsys' "Recommended Methodology" download.

We are not implementing scan chains in this exploratory build.

Once a user has customized the flow for an individual block's
requirements, fully building the block from RTL to a final physical
implementation timed in a signoff-quality is a single "make" command
much like the compilation of any software.

Target technology is a prevous-generation commodity process available
on the open market which we used for a recent chip design at Centaur.
Stdcell and hard macro (custom block) libraries are Centaur-developed
- our own layout and timing characterization.


Build steps for new RTL:
------------------------------------------------------------------------
1) Initial mapping RTL to stdcells
2) Create floorplan
3) Map RTL to stdcells with floorplan
4) Import stdcells into physical implementation tool
5) Place stdcells and optimize
6) Build clock tree, route & optimize it
7) Route all nets
8) Extract parasitics and run sign-off timing tool

Initial DC-NXT compile with NO FLOORPLAN and a loose cycletime to
create technology-mapped gates and initial timing feedback. Optionally,
a second pass without a floorplan could be run at a tighter cycletime
since a tighter cycletime might require a larger stdcell area. The goal
here is to have gates to use in creation of a floorplan.

A floorplan is created based on initial gates and block
requirements. Since we don't know yet what dimensions the block will
```

be when it is instantiated in a larger framework, we can build roughly square with inputs on the left and outputs on the right. We'll also keep the I/O timing constraints loose so the placement of I/O pins won't be a problem for timing or routing.

DC-NXT is run again to map RTL to gates, but with a floorplan this time. We run in SPG mode (Synopsys Physical Guidance?) topgraphic mode so the compiler understands the physical world - e.g., placement, metal layer parasitics. An initial compile is followed by two incremental compiles - more compiles might produce better final timing results but two incremental compiles is often sufficient.

DC-NXT can bank registers, but in this build we're banking registers using ICC2 during placement. Our library supports two to four registers banked into a single standard cell (stdcell aka leaf cell).

In DC-NXT we permit register replication, boundary optimization (optimizing logic between hierarchicies i.e. RTL modules), sequential output inversion (inverting nets passing between hierarchies), automated layer assignment (choose the likely metal layer critical nets will be routed on). During exploratory work we do not permit DC-NXT to dissolve hierarchical boundaries, to help with understanding the reported paths. In a large design we can often improve the final timing by ~5% by permitting dissolution of boundaries (aka auto-ungrouping).

Importing gates from DC-NXT into the physical implementation tool (ICC2) is a simple and fast step that requires no user optimization.

ICC2 places and optimizes the gates for timing, routability, and power. We discard the placement coming from DC because ICC2 generally does a better job restarting with its own coarse placement alogrithm. We run a two-pass placement, allow ICC2 to assign "NDRs" (non-default rules i.e., choosing upper layers of metal and/or wider nets), and implement register banking. We also enable CCD (concurrent clock & data) optimization which is free to push/pull register and macro clocks foward or backwards to optimize timing.

ICC2 builds a clock tree, routes it, and optimizes it. There is almost no user customization here although we also allow CCD optimization here.

ICC2 routes and optimizes the remaining non-clock nets. We run multiple optimization passes to improve timing and route quality, allow CCD optimization in early passes but not in later passes.

Finally the ICC2 output is extracted in STAR to provide actual RCs (including cross-coupling between routes) which are then fed into Primetime for analysis at our typical setup & hold corners. For sign-off we use many additional corners but typical setup & hold are considered adequate for exploring new RTL designs.

APPENDIX E. BUILD RESULTS

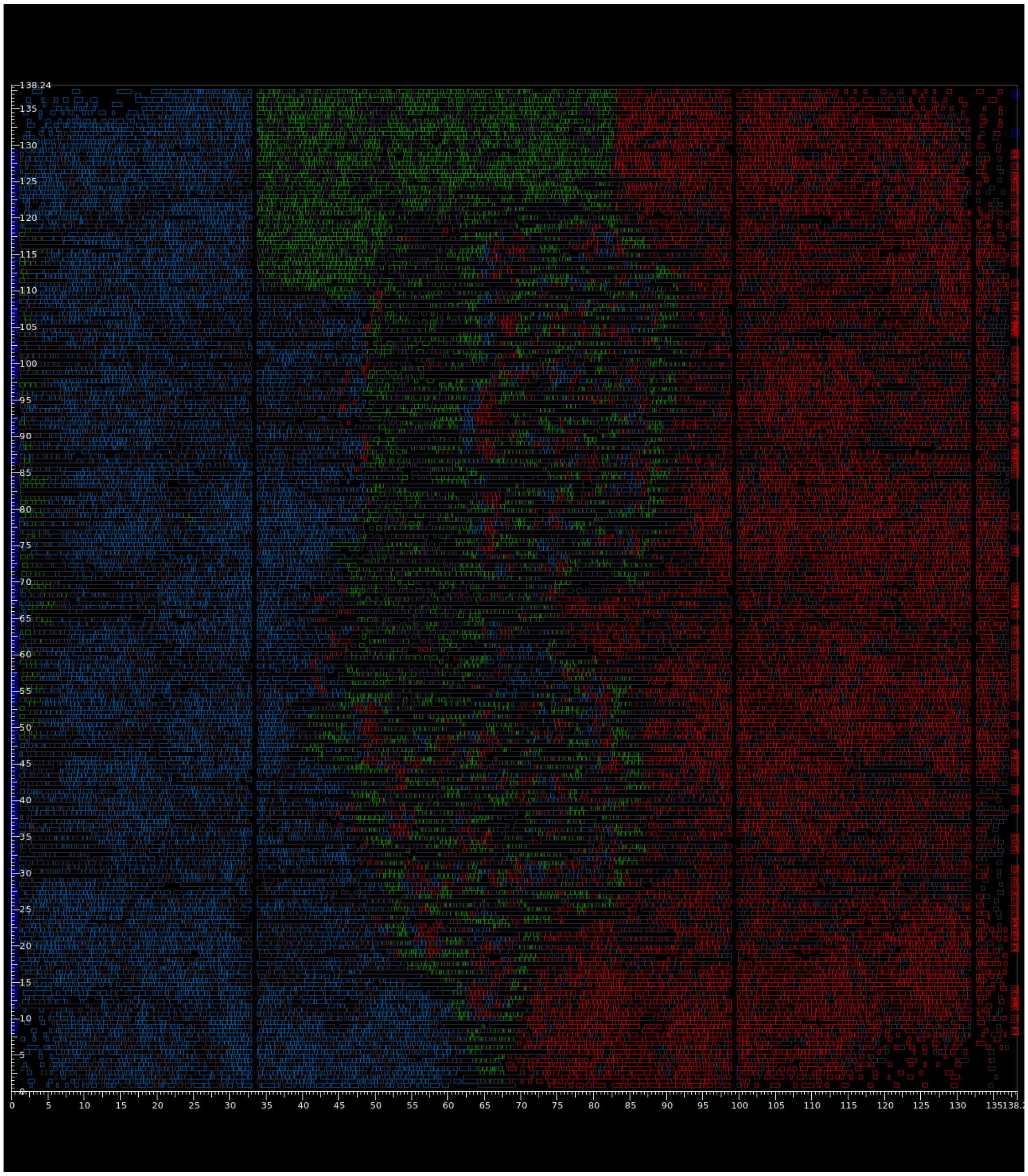
Adapted from: Jim Donahue, Centaur Technologies [9].

```
Physical Builds
----------------------
Three versions of the code (AES, Xoodyak 1 round/clock, Xoodyak 3
rounds/clock) were built by Jim Donahue at Centaur Technology Inc. In order to identify any
differences due to technology, builds were done in both a 16nm technology and a 5nm
technology. The build “recipes”
(details in Appensix Q) were basically the same as used by Centaur for other internal
designs.

Within each technology, the only thing changed across the builds for a particular code
version were the target clock frequency (and a few floorplan size changes thathad no affect
on the results). Across the three code versions, fourteen build variations were done in
16nm, and twelve in 5nm. The intent of the various build configurations were to find (1)
the fastest realistic frequency for a design, and (2) comparisons across the designs at the
same (roughly) frequency. Appendix y summarizes the data from all builds and the following
table show the versions that of particular interest for this study.

-------------------------------------------------
target = Target GHz for build
x y  = build die area, not necessarily indicative of real are actually needed
real freq = the mhz the build
produced area = total area of standard cells used without wiring, etc.
util = cell utilization % vs available die area, not critical until above 70%
leak = static leakage in mW, rule-of-thumb cutoff is around 1mW for small circuits like Xoo
total power = leakage + dynamic.
leakage is telling us what type of cells are being used.

X == Selected runs for detailed comparison
------------------------------------------------
Tabulated results on next page
```

table---------------------------------------

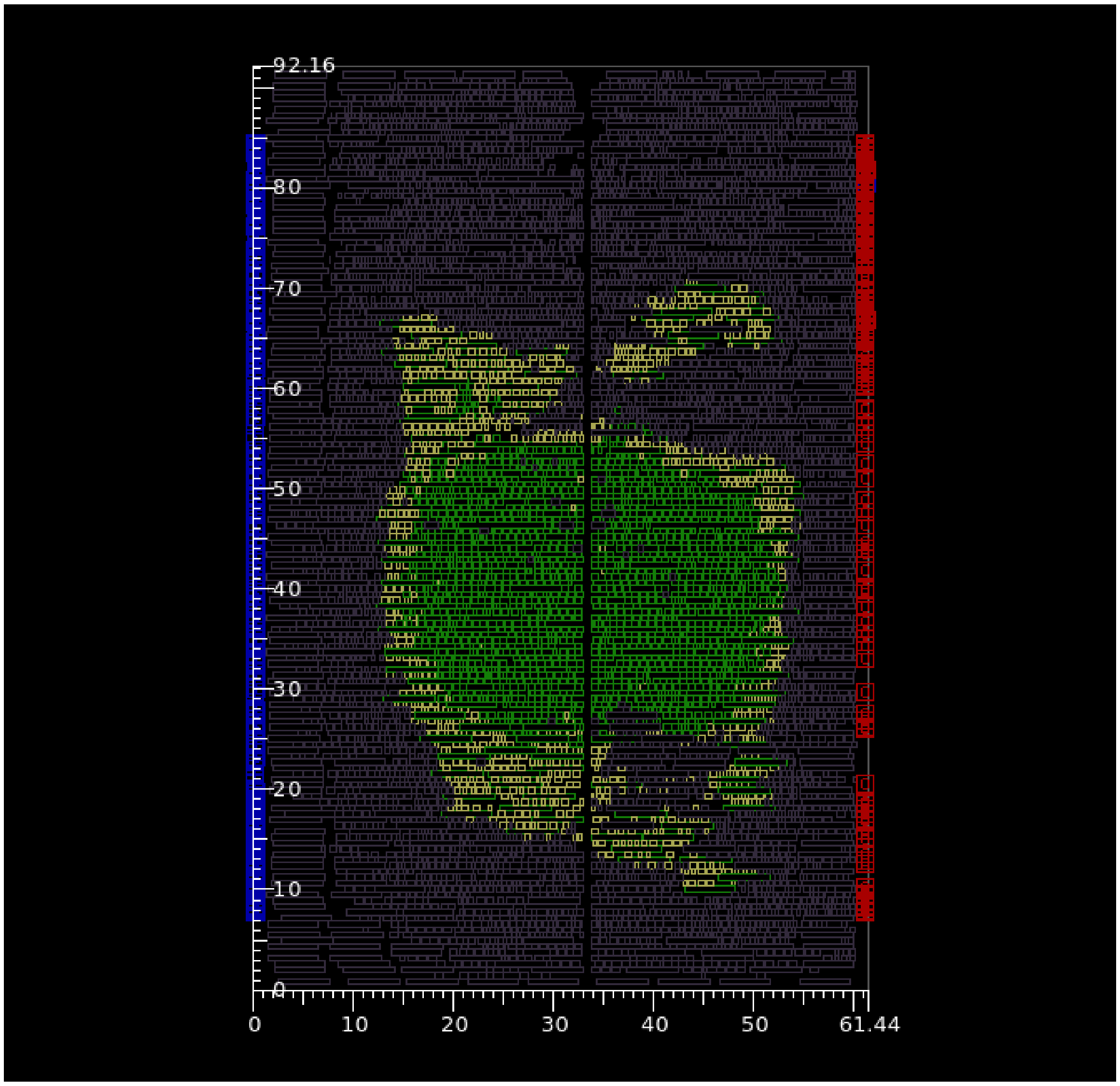
exp	target	x y	real freq	area	util	leak	X
aes							
a.03d	2.00	138 138	2.06	7073	38	0.46	
a.03c	2.35	138 138	2.42	7432	40	0.81	X
a.03b	3.64	138 138	3.56	10365	55	5.34	
a.03g	2.67	138 138	2.69	7782	42	1.24	
a.03h	2.86	138 138	2.91	8147	44	1.93	
a.03i	3.33	138 138	3.37	9307	50	3.71	
a.03e	4.00	138 138	3.60	11313	61	7.47	
a.03f	5.00	138 138	3.82	12025	64	9.30	
a.5n03f	2.00	64 64	2.07	2171	55	0.20	X
a.5n03c	2.86	64 64	2.95	2342	59	0.36	
a.5n03d	4.00	64 64	4.00	2782	71	0.83	X
a.5n03e	5.00	80 80	4.33	3736	60	1.81	
xoo 1 perm							
x.07a	2.11	61 92	2.12	3380	63	0.23	
x.07b	2.35	61 92	2.40	3298	61	0.23	X
x.07c	2.86	122 107	2.89	3508	27	0.43	
x.07d	3.33	122 107	3.38	4162	33	0.78	X
x.07e	4.00	122 107	4.04	4722	37	1.64	
x.07h	5.00	92 107	4.18	5769	61	3.58	
x.5n05a	2.86	45 45	3.05	1173	61	0.22	
x.5n05c	3.33	45 45	3.42	1180	62	0.25	
x.5n05d	4.00	45 45	4.27	1368	71	0.33	
x.5n05e	5.00	45 45	5.03	1560	81	0.44	
x.5n05g	5.00	60 60	5.09	2060	60	0.77	X
x00 3-perm							
x.08a	2.11	123 107	2.15	6347	50	2.51	X
x.08b	2.35	123 107	2.42	7712	60	4.65	
x.08c	2.67	123 107	2.33	9429	74	6.41	
x.5n06b	2.50	85 85	2.60	1860	26	0.64	
x.5n06a	2.86	85 85	2.92	2212	32	0.86	X
x.5n06c	3.33	85 85	3.07	2583	37	1.30	

APPENDIX F. SAMPLE BUILD IMAGES



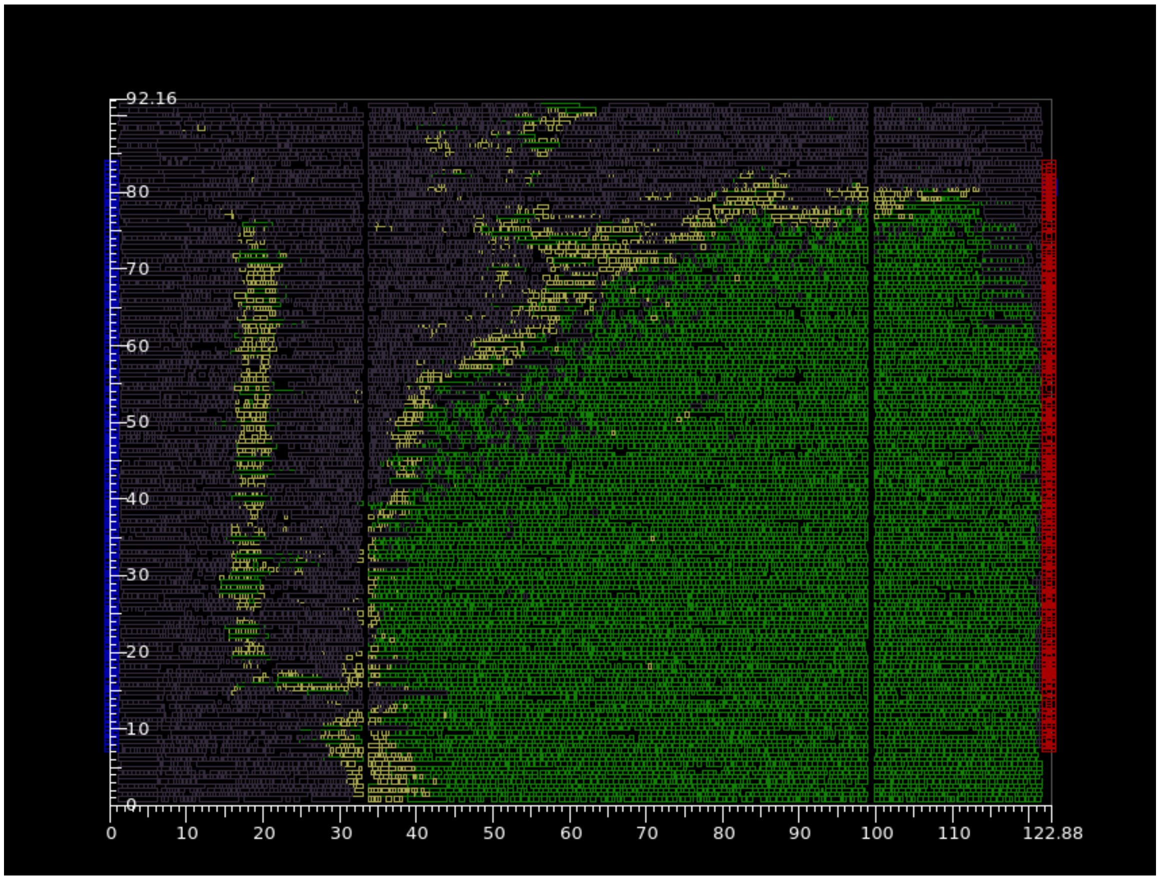
(a) Blue cells - *Decrypt* logic. (b) Green cells - *Encrypt* logic. (c) Red cells - *KeyExpansion*

Figure 24. A-03c Die Space. Source: [9].



(a) Green cells - permute round logic. (b) White cells - permute support logic. (c) Gray cells - All other combinational logic. (d) Blue cells - input ports. (e) Red cells - output ports.

Figure 25. X-07b Die Space. Source: [9]



(a) Green cells - permute round logic. (b) White cells - permute support logic. (c) Gray cells - All other combinational logic. (d) Blue cells - input ports. (e) Red cells - output ports. [9]

Figure 26. X-08a Die Space. Source: [9]

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] *Advanced Encryption Standard (AES),* NIST FIPS - 197, National Institute of Standards and Technology, 2001

[2] D. McGrew, “An interface and algorithms for authenticated encryption,” IETF Datatracker, Jan. 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5116#section-1

[3] M. Dworkin, “Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC,” National Institute of Standards and Technology , Gaithersburg, MD, USA, NIST Special Publication 800–38D, 2007, pp. 1–39.

[4] Computer Security Division, K. A. McKay, L. Bassham, M. S. Turan, and N. Mouha, NISTIR 8114, Report on Lightweight Cryptography. National Institute of Standards and Technology, pp. 1–27. https://doi.org/10.6028/NIST.IR.8114

[5] J. Daemen, S. Hoffert, M. Peters, G. Van Assche, and R. Van Keer, “Xoodyak, a lightweight cryptographic scheme,” Lightweight Cryptography Round 2 Candidates, 29-Mar-2019. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Xoodyak-spec-round2.pdf

[6] Vivado Design Suite, 2020.2. San Jose, CA, USA: Xilinx, April 2012. [Online]. Available: https://www.xilinx.com/support/download.html, Accessed on: 2019–2021.

[7] David Hill, MathWorks. 2021. Advanced Encryption Standard (AES)-128,192, 256, ver. 1.0.4 [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/73412-advanced-encryption-standard-aes-128-192-256

[8] S. B. Palmer, Github. 2021. Xoocycle, [ONLINE], Available: https://github.com/sbp/xoocycle

[9] J. Donahue, Build Results 21OCT, Centaur Technologies. Unpublished.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California